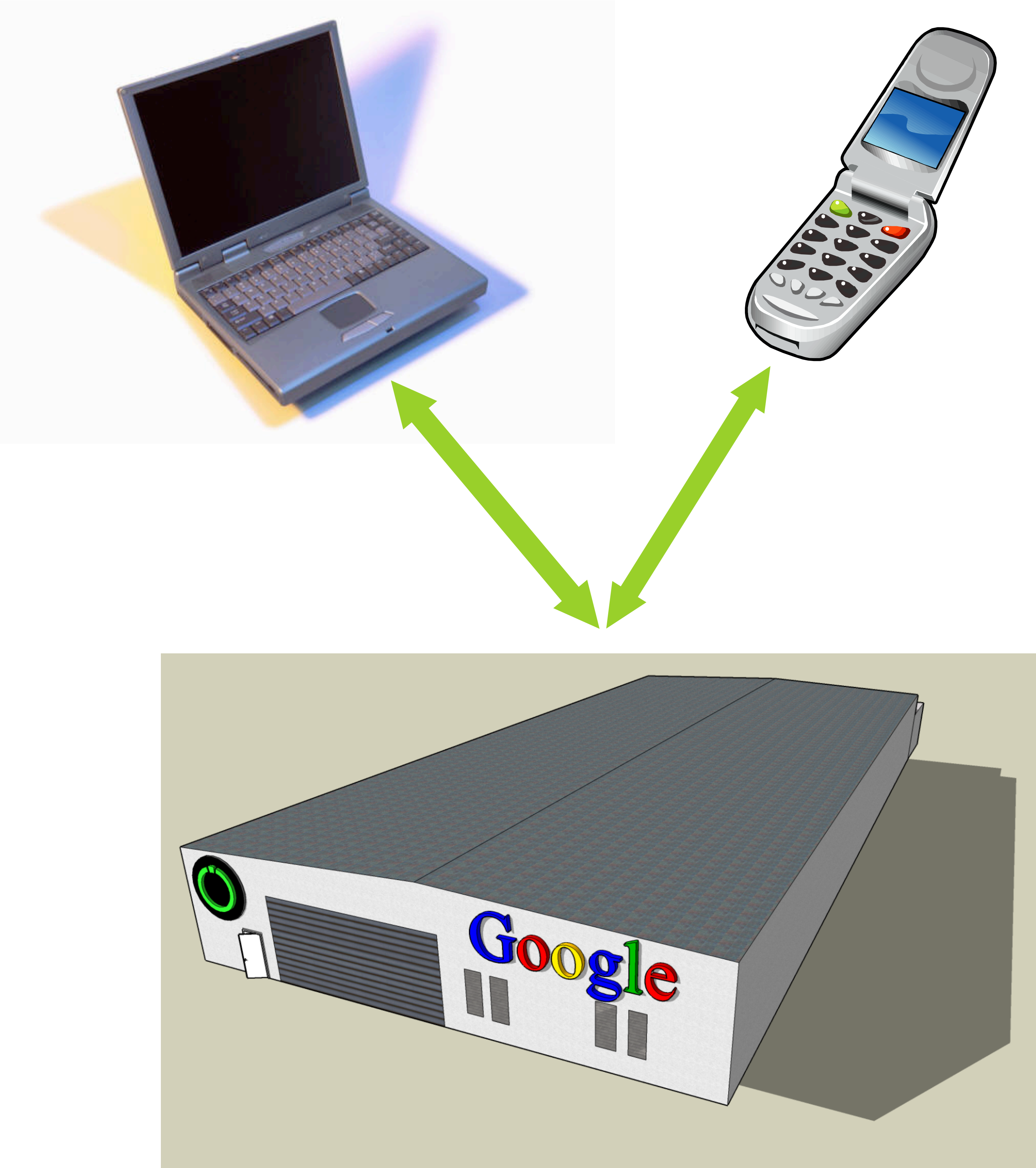
Designs, Lessons and Advice from Building Large Distributed Systems

Jeff Dean
Google Fellow
jeff@google.com

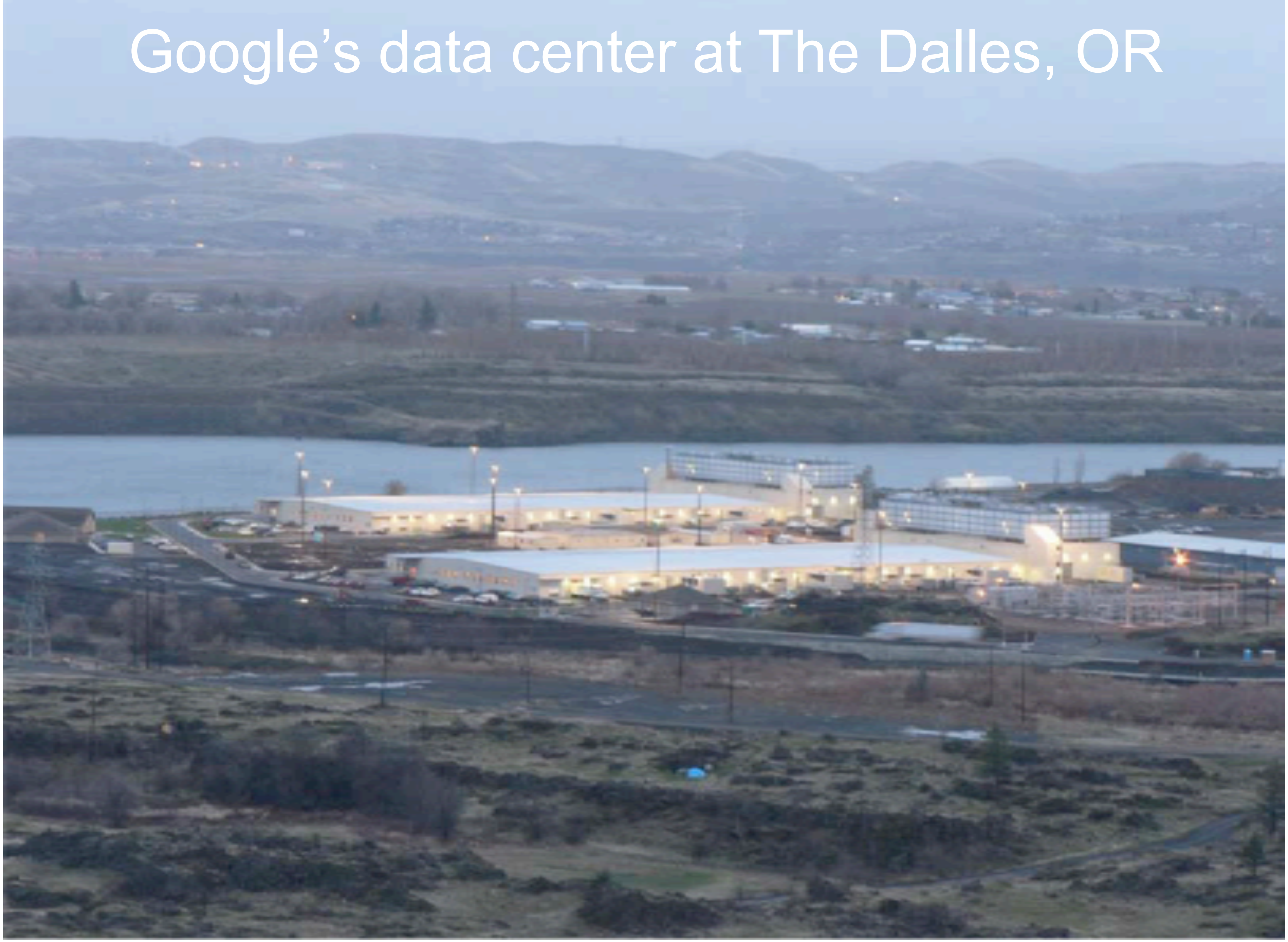
Computing shifting to really small and really big devices



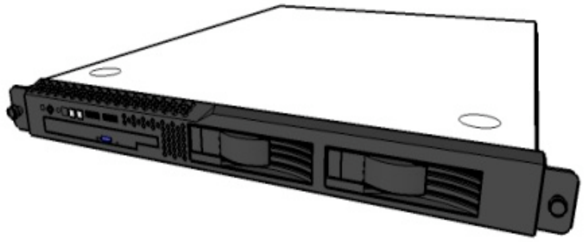
UI-centric devices

Large consolidated computing farms

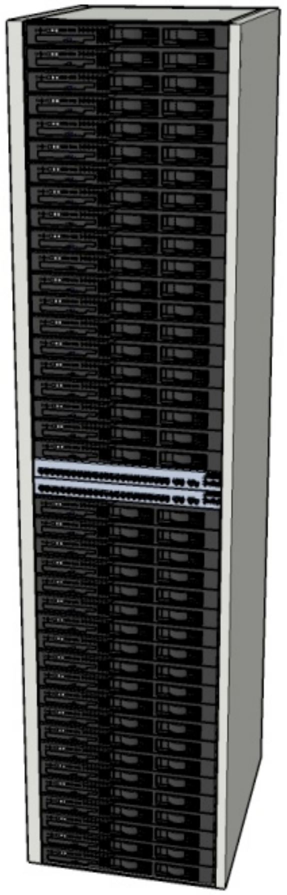
Google’s data center at The Dalles, OR



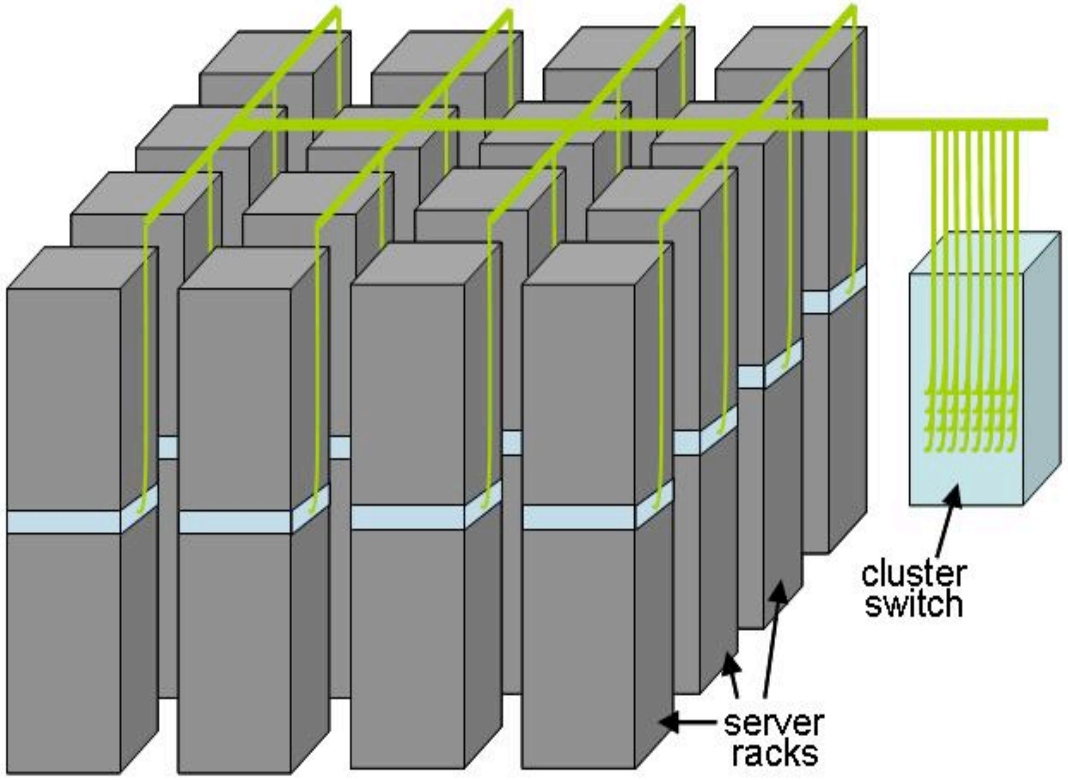
The Machinery



Servers
- CPUs
- DRAM
- Disks

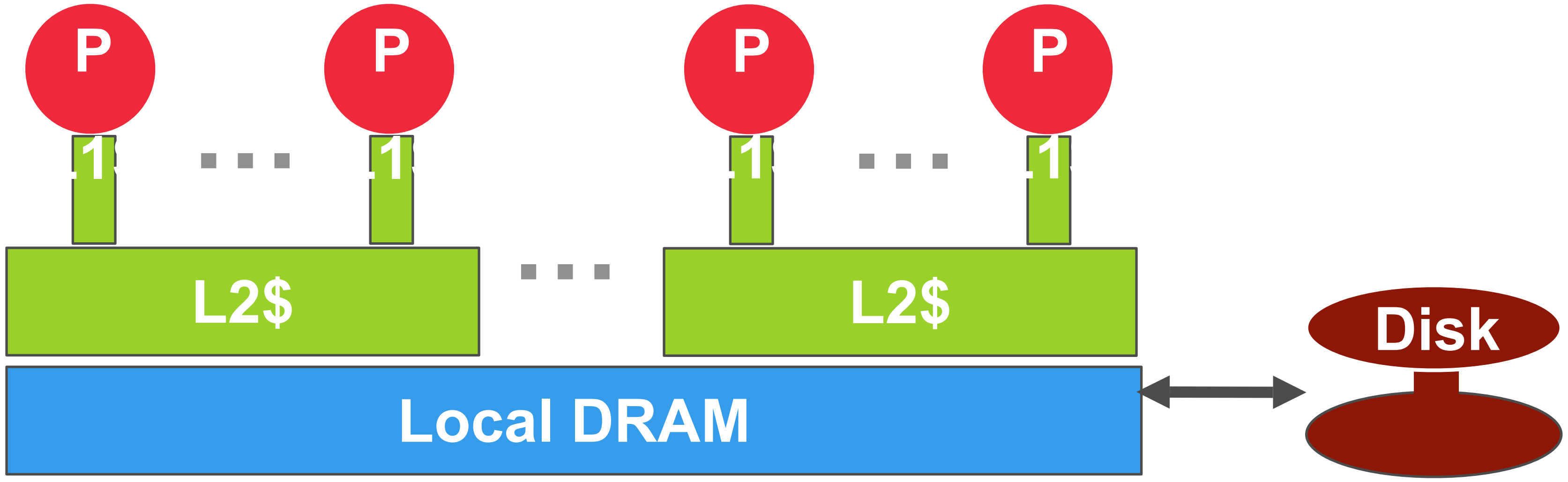


Racks
- 40-80 servers
- Ethernet switch



Clusters

Architectural view of the storage hierarchy

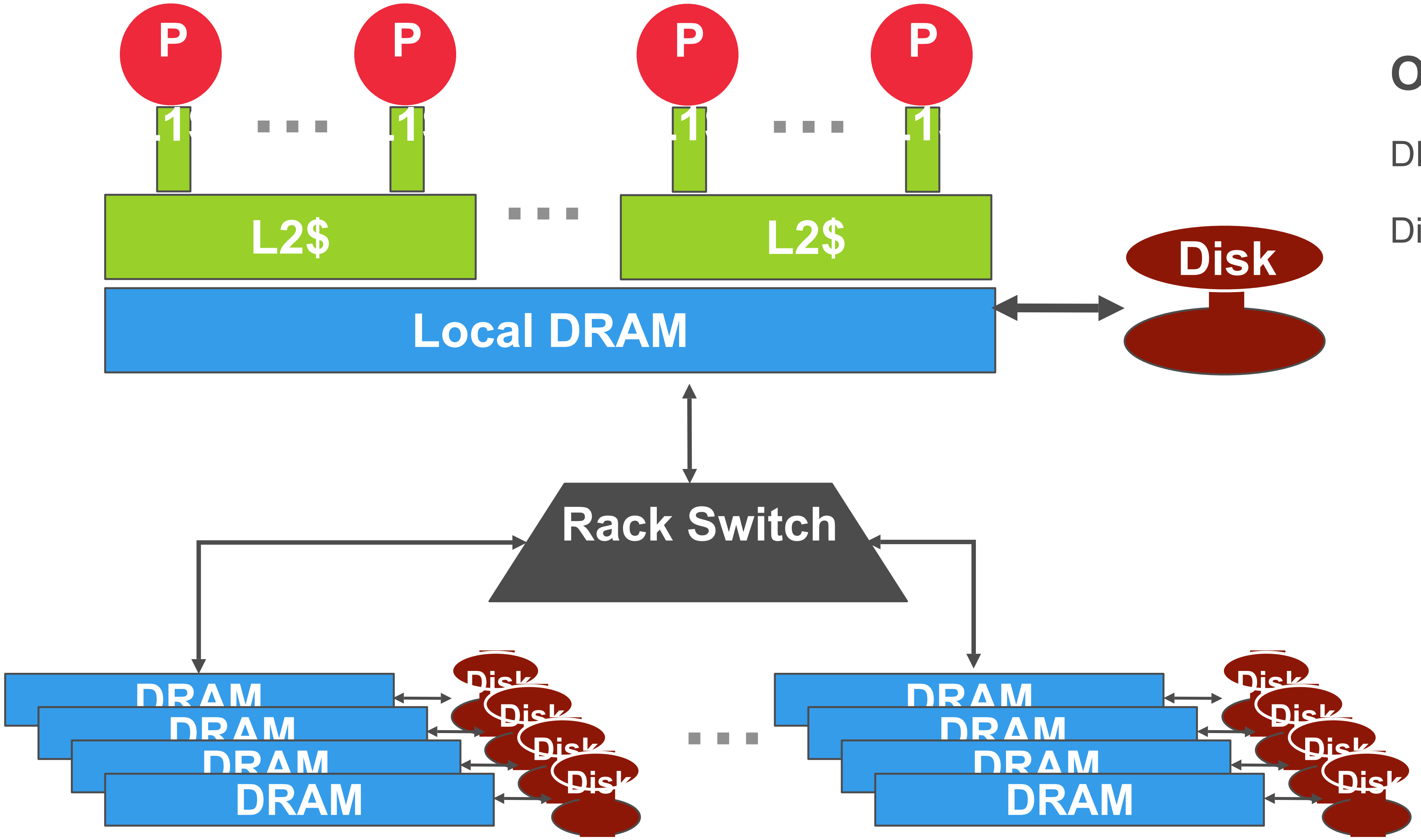


One server

DRAM: 16GB, 100ns, 20GB/s

Disk: 2TB, 10ms, 200MB/s

Architectural view of the storage hierarchy



One server

DRAM: 16GB, 100ns, 20GB/s

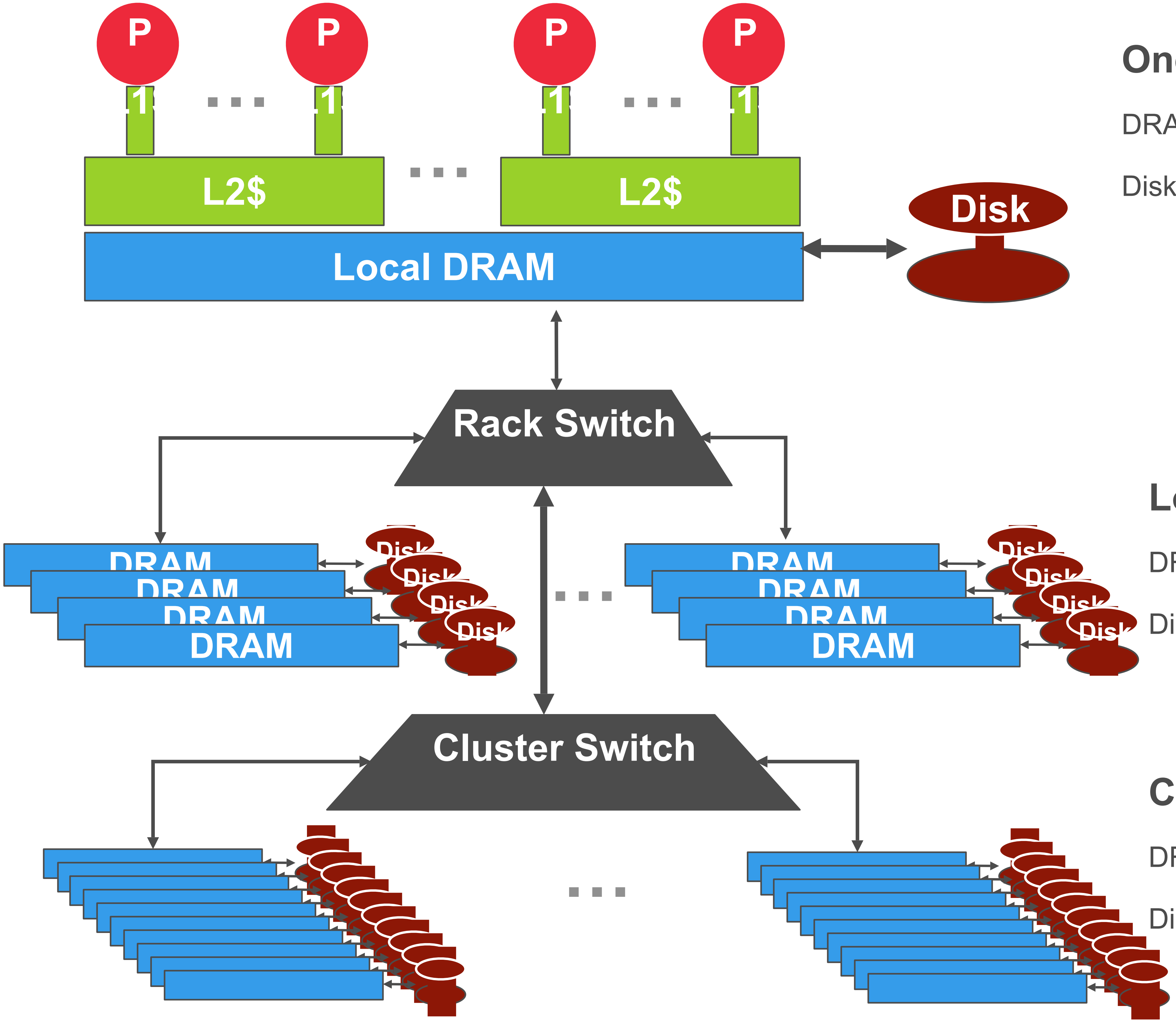
Disk: 2TB, 10ms, 200MB/s

Local rack (80 servers)

DRAM: 1TB, 300us, 100MB/s

Disk: 160TB, 11ms, 100MB/s

Architectural view of the storage hierarchy



One server

DRAM: 16GB, 100ns, 20GB/s

Disk: 2TB, 10ms, 200MB/s

Local rack (80 servers)

DRAM: 1TB, 300us, 100MB/s

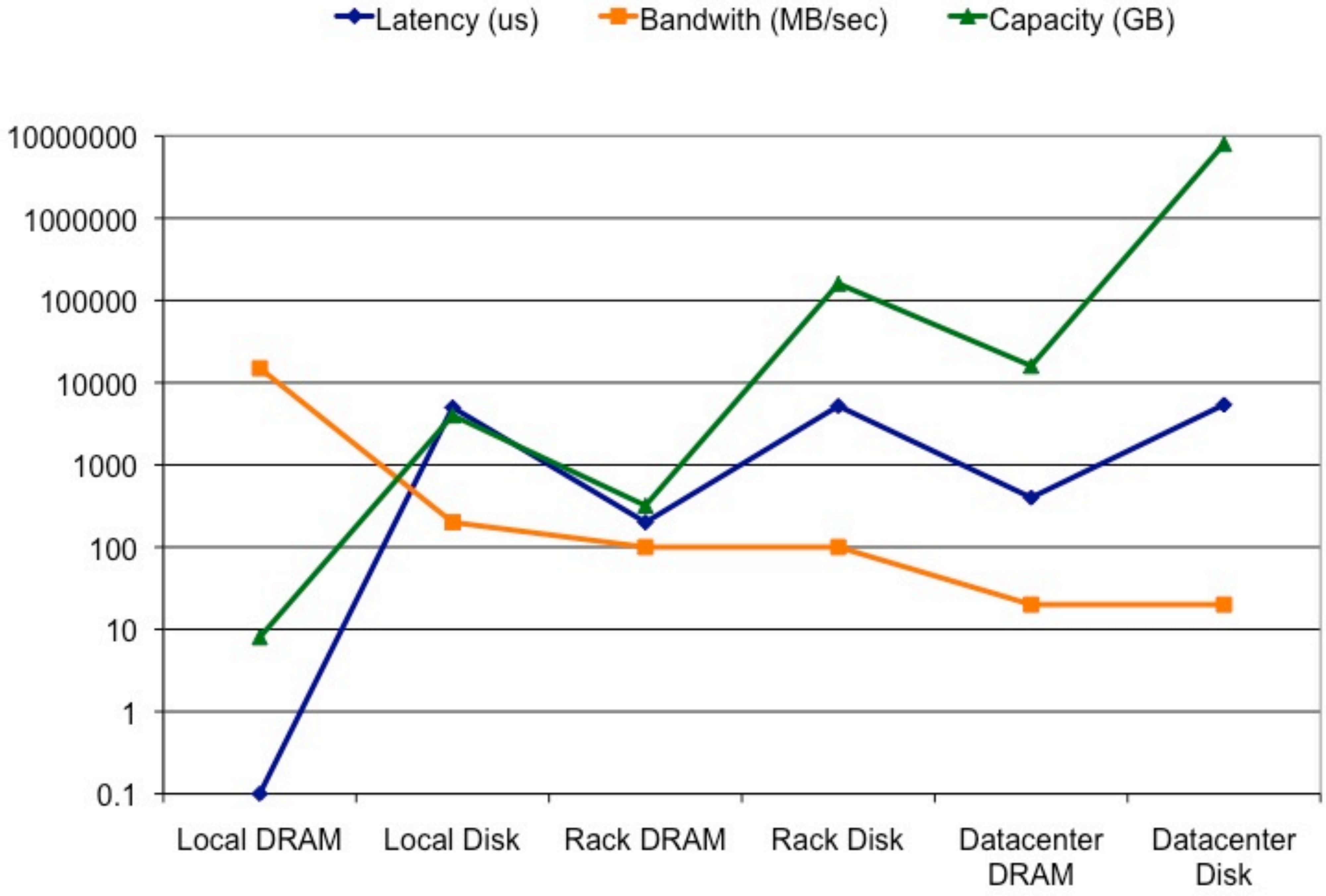
Disk: 160TB, 11ms, 100MB/s

Cluster (30+ racks)

DRAM: 30TB, 500us, 10MB/s

Disk: 4.80PB, 12ms, 10MB/s

Storage hierarchy: a different view



A bumpy ride that has been getting bumpier over time

Reliability & Availability

- Things will crash. Deal with it!
  - Assume you could start with super reliable servers (MTBF of 30 years)
  - Build computing system with 10 thousand of those
  - ***Watch one fail per day***

- Fault-tolerant software is inevitable

- Typical yearly flakiness metrics
  - 1-5% of your disk drives will die
  - Servers will crash at least twice (2-4% failure rate)

The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)

~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)

~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packetloss)

~8 network maintenances (4 might cause ~30-minute random connectivity losses)

~12 router reloads (takes out DNS and external vips for a couple minutes)

~3 router failures (have to immediately pull traffic for an hour)

~dozens of minor 30-second blips for dns

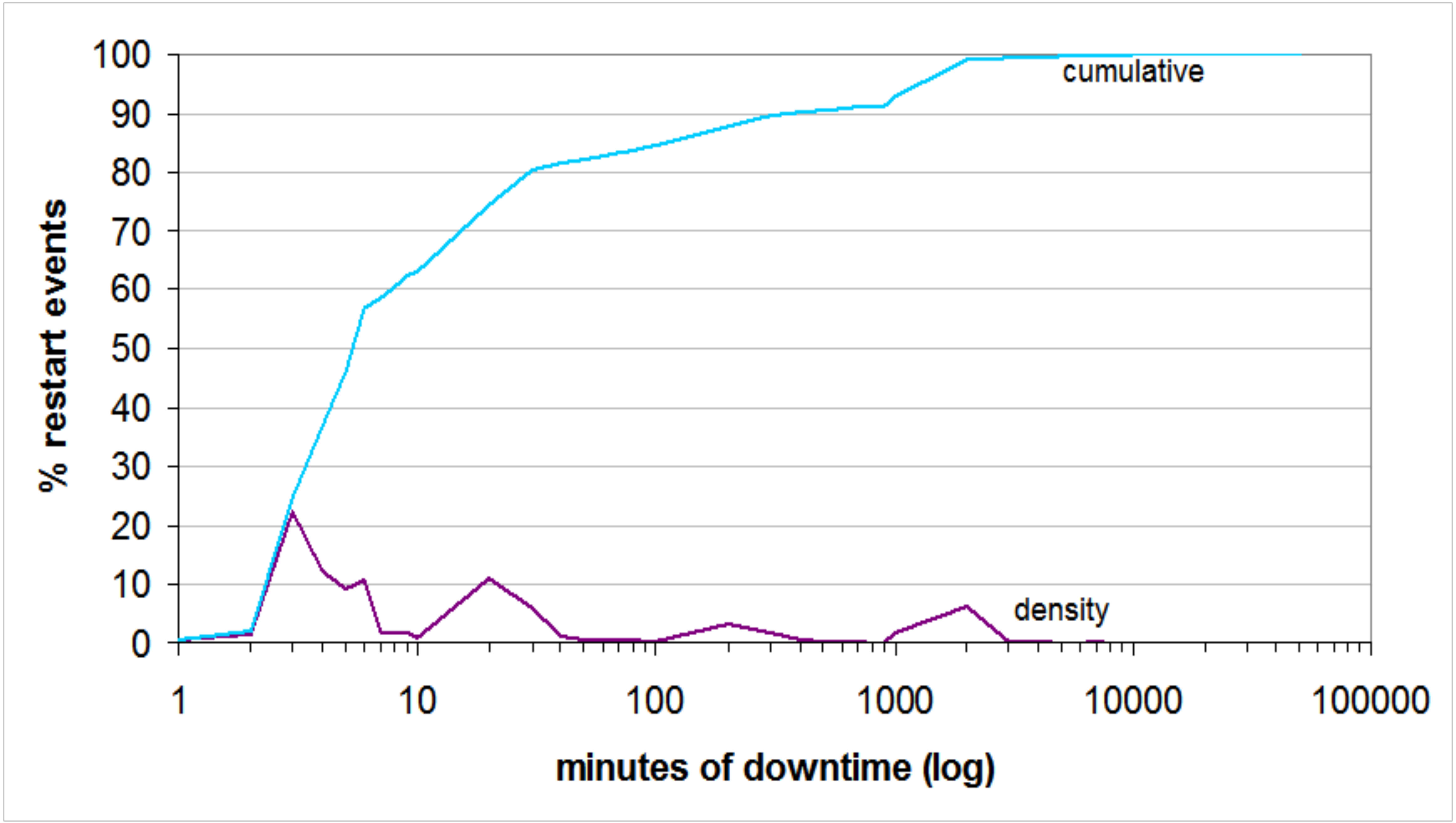
~1000 individual machine failures

~thousands of hard drive failures

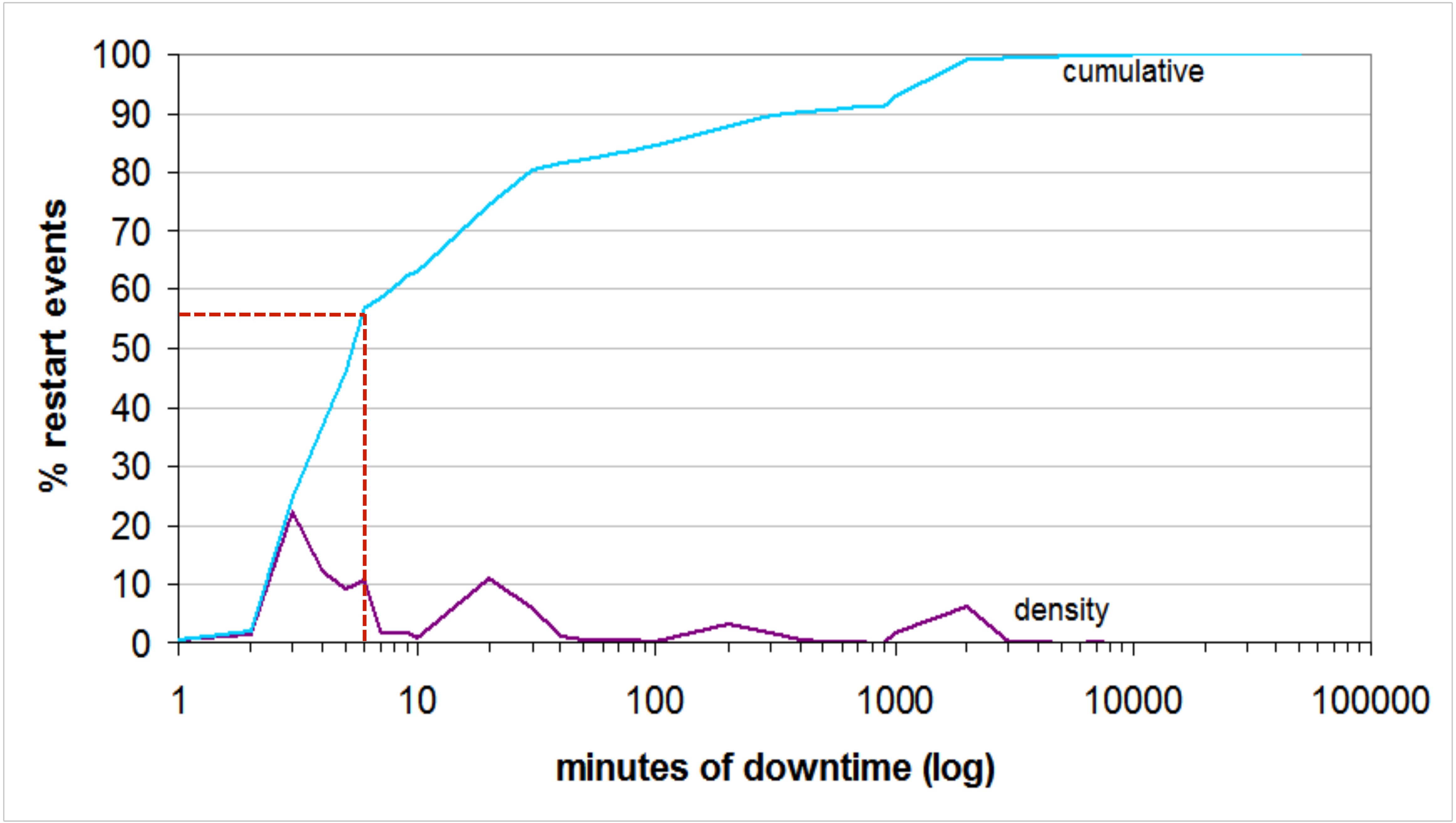
slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

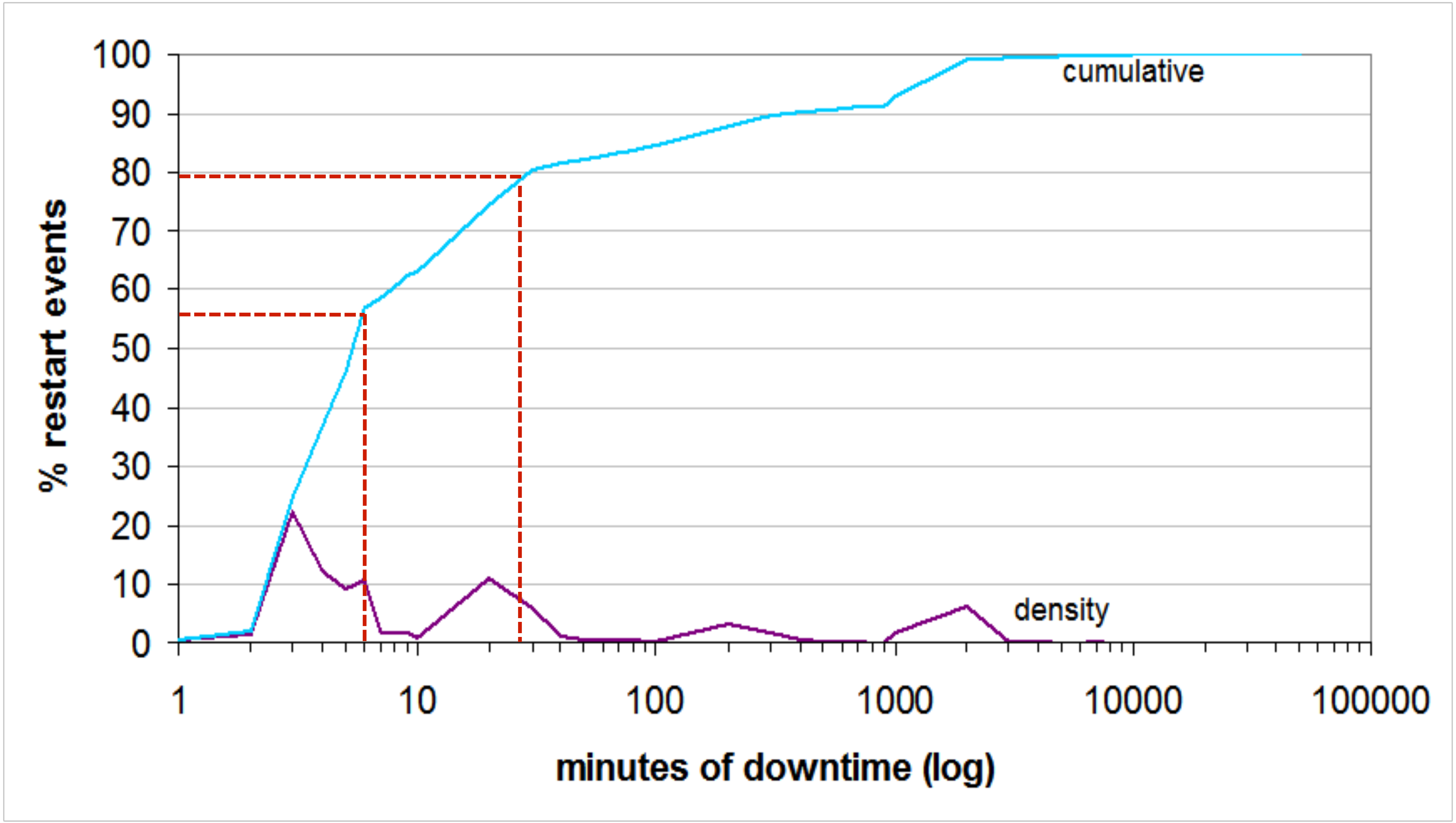
Understanding downtime behavior matters



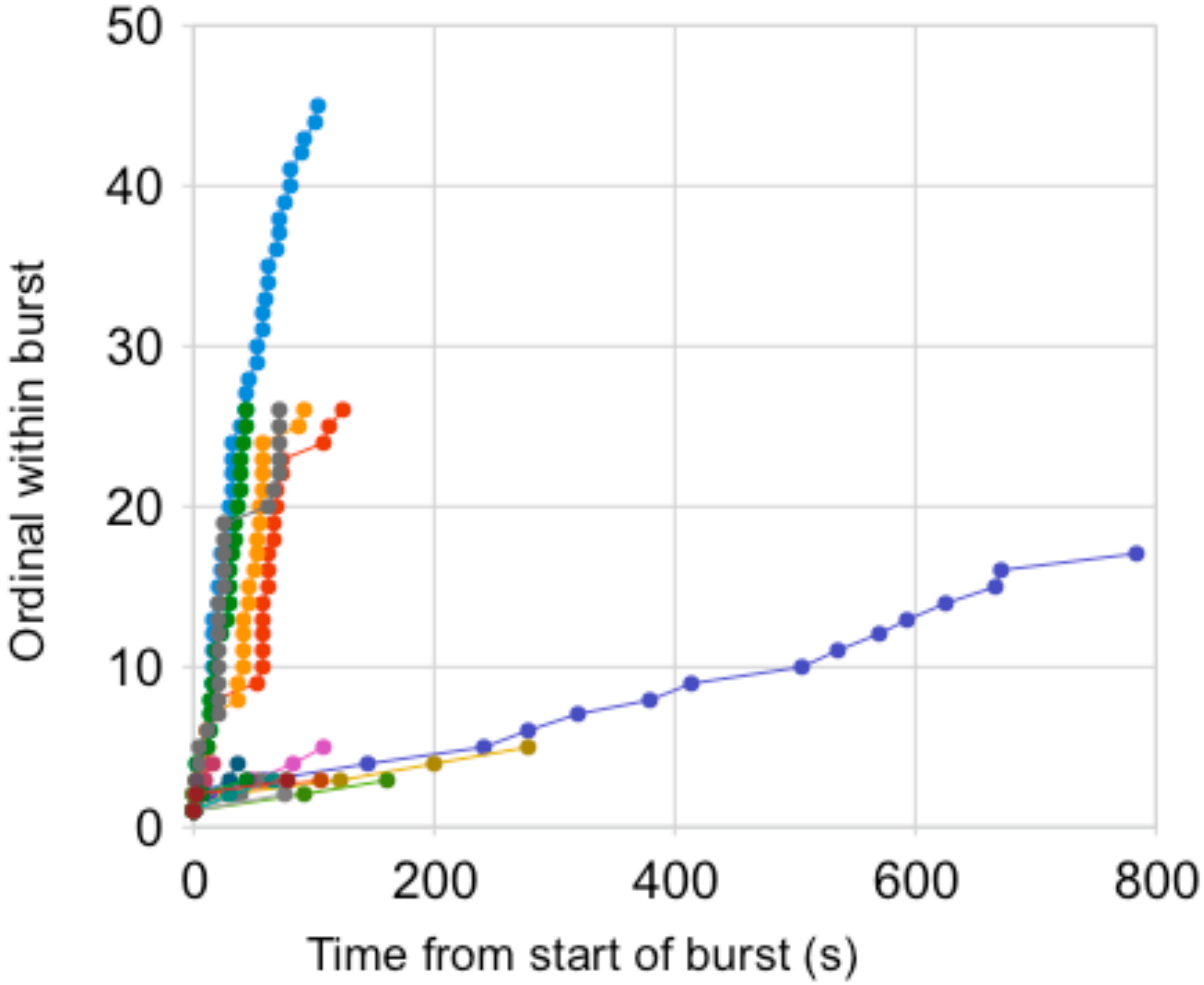
Understanding downtime behavior matters



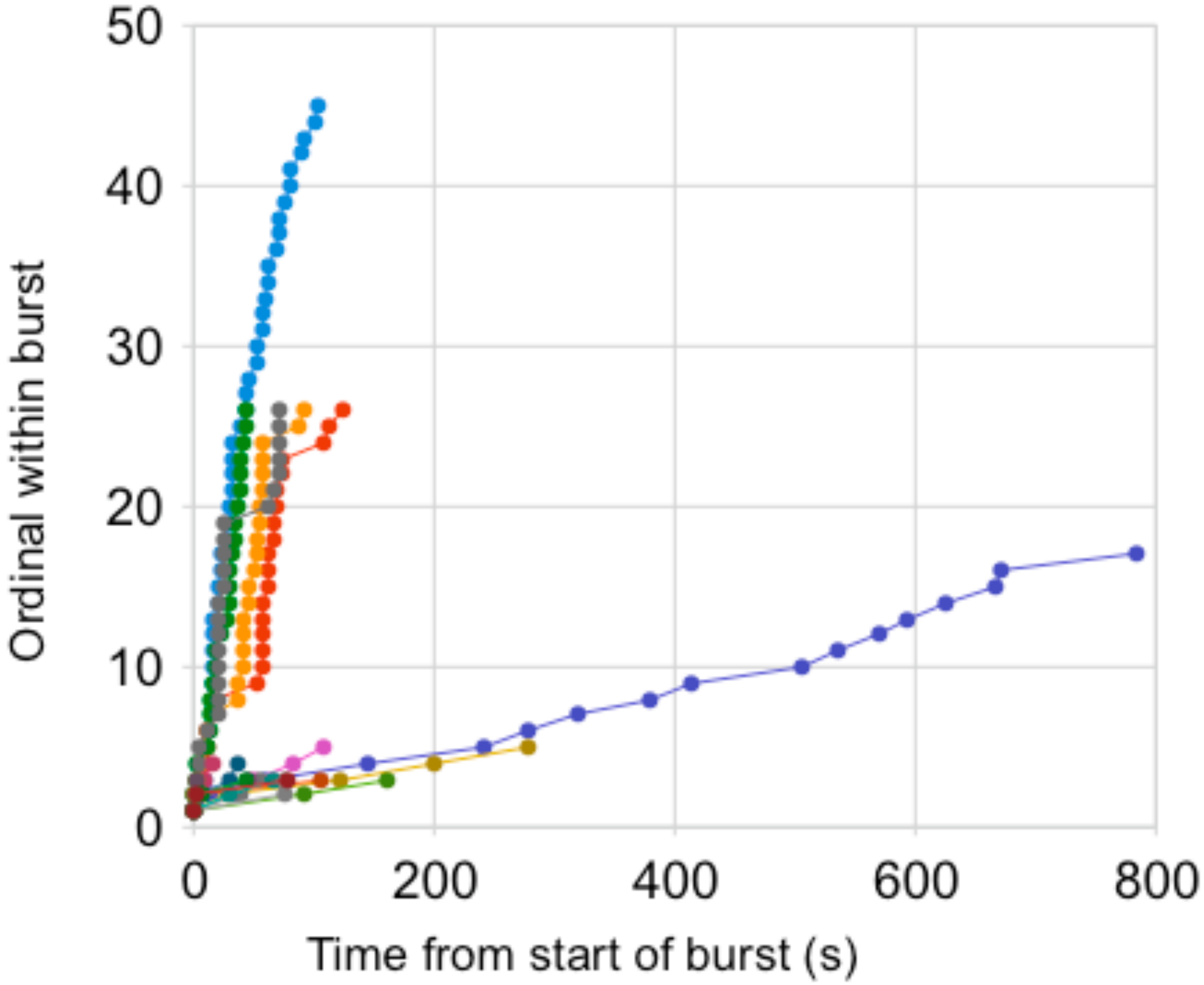
Understanding downtime behavior matters



Understanding fault statistics matters



Understanding fault statistics matters

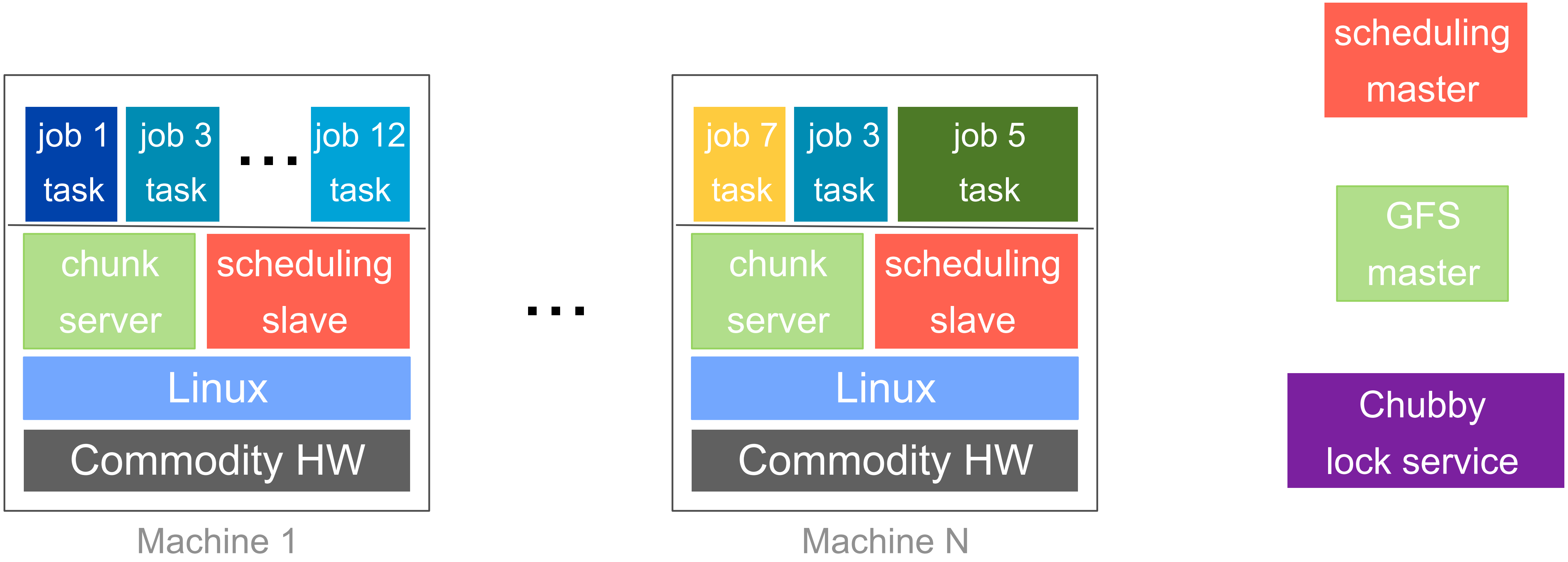


Can we expect faults to be independent or correlated?

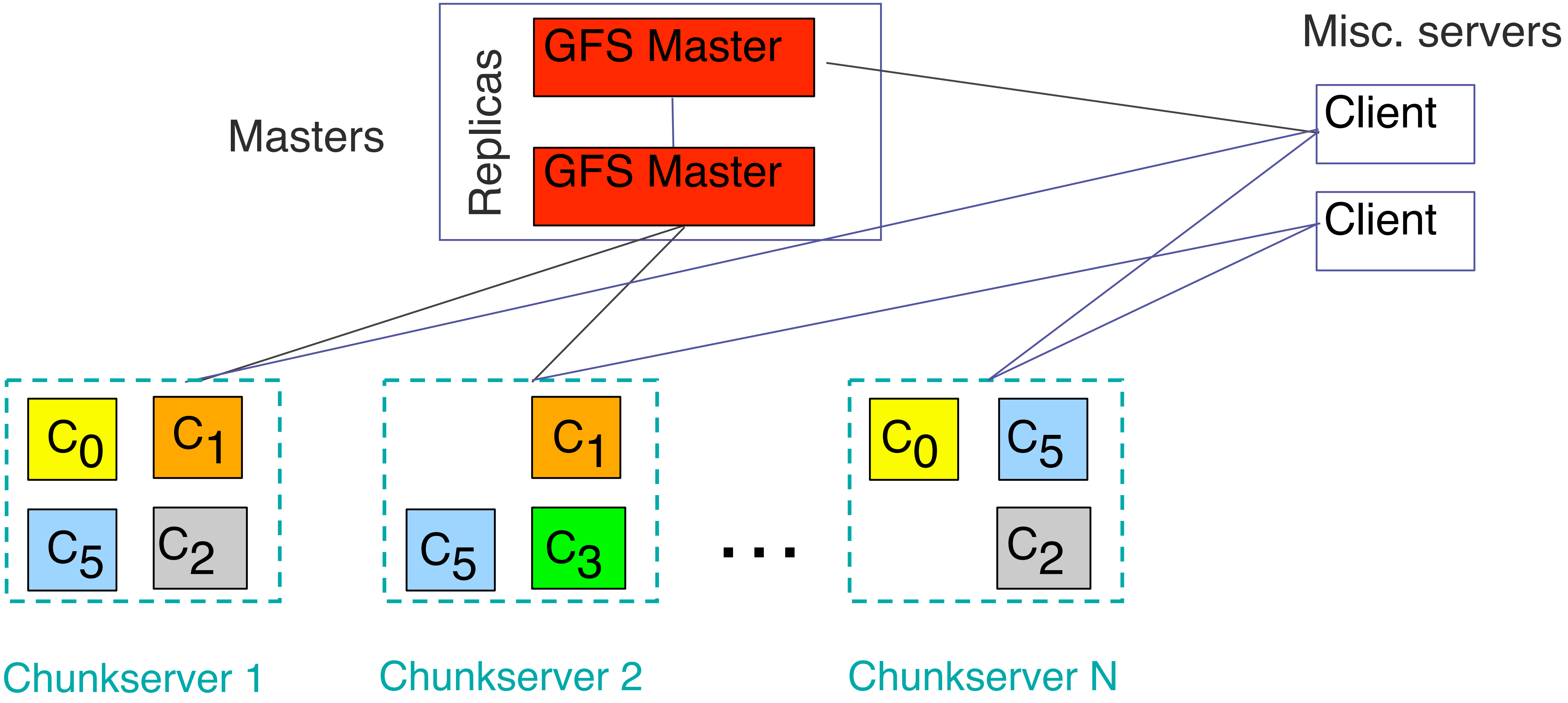
Are there common failure patterns we should program around?

Google Cluster Environment

- Cluster is 1000s of machines, typically one or handful of configurations
- File system (GFS) + Cluster scheduling system are core services
- Typically 100s to 1000s of active jobs (some w/1 task, some w/1000s)
    - mix of batch and low-latency, user-facing production jobs



GFS Design



Misc. servers

Client

Client

Masters

Replicas

GFS Master

GFS Master

C_0 C_1
C_5 C_2

C_1
C_5 C_3

...

C_0 C_5
C_2

Chunkserver 1

Chunkserver 2

Chunkserver N

- Master manages metadata
- Data transfers happen directly between clients/ chunkservers
- Files broken into chunks (typically 64 MB)

GFS Usage @ Google

- 200+ clusters
- Many clusters of 1000s of machines
- Pools of 1000s of clients
- 4+ PB Filesystems
- 40 GB/s read/write load
  - (in the presence of frequent HW failures)

Google: Most Systems are Distributed Systems

- Distributed systems are a must:
  - data, request volume or both are too large for single machine
    - careful design about how to partition problems
    - need high capacity systems even within a single datacenter
  - multiple datacenters, all around the world
    - almost all products deployed in multiple locations
  - services used heavily even internally
    - a web search touches 50+ separate services, 1000s machines

Many Internal Services

- Simpler from a software engineering standpoint
  - few dependencies, clearly specified (Protocol Buffers)
  - easy to test new versions of individual services
  - ability to run lots of experiments

- Development cycles largely decoupled
  - lots of benefits: small teams can work independently
  - easier to have many engineering offices around the world

Protocol Buffers

- Good protocol description language is vital
- Desired attributes:
  - self-describing, multiple language support
  - efficient to encode/decode (200+ MB/s), compact serialized form

Our solution: Protocol Buffers (in active use since 2000)

```
message SearchResult {
  required int32 estimated_results = 1;  // (1 is the tag number)
  optional string error_message = 2;
  repeated group Result = 3 {
    required float score = 4;
    required fixed64 docid = 5;
    optional message<WebResultDetails> = 6;
    …
  }
};
```

Protocol Buffers (cont)

- Automatically generated language wrappers
- Graceful client and server upgrades
  - systems ignore tags they don't understand, but pass the information through (no need to upgrade intermediate servers)
- Serialization/deserialization
  - high performance (200+ MB/s encode/decode)
  - fairly compact (uses variable length encodings)
  - format used to store data persistently (not just for RPCs)
- Also allow service specifications:

```
service Search {
  rpc DoSearch(SearchRequest) returns (SearchResponse);
  rpc DoSnippets(SnippetRequest) returns
 (SnippetResponse);
  rpc Ping(EmptyMessage) returns (EmptyMessage) {
    { protocol=udp; };
};
```

- Open source version: http://code.google.com/p/protobuf/

Designing Efficient Systems

Given a basic problem definition, how do you choose the "best" solution?

- Best could be simplest, highest performance, easiest to extend, etc.

Important skill: ability to estimate performance of a system design

  - without actually having to build it!

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Back of the Envelope Calculations

How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

```
30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms
```

Back of the Envelope Calculations

How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

```
30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms
```

Design 2: Issue reads in parallel:

```
10 ms/seek + 256K read / 30 MB/s = 18 ms
```

(Ignores variance, so really more like 30-60 ms, probably)

Back of the Envelope Calculations

How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

`30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms`

Design 2: Issue reads in parallel:

`10 ms/seek + 256K read / 30 MB/s = 18 ms`

(Ignores variance, so really more like 30-60 ms, probably)

Lots of variations:

- caching (single images? whole sets of thumbnails?)
- pre-computing thumbnails
- …

Back of the envelope helps identify most promising…

Write Microbenchmarks!

- Great to understand performance
  - Builds intuition for back-of-the-envelope calculations
- Reduces cycle time to test performance improvements

```
Benchmark                      Time(ns)  CPU(ns) Iterations
BM_VarintLength32/0                   2        2  291666666
BM_VarintLength32Old/0                5        5  124660869
BM_VarintLength64/0                   8        8   89600000
BM_VarintLength64Old/0               25       24   42164705
BM_VarintEncode32/0                   7        7   80000000
BM_VarintEncode64/0                  18       16   39822222
BM_VarintEncode64Old/0               24       22   31165217
```

Know Your Basic Building Blocks

Core language libraries, basic data structures,
protocol buffers, GFS, BigTable,
indexing systems, MySQL, MapReduce, …

Not just their interfaces, but understand their implementations (at least at a high level)

If you don’t know what’s going on, you can’t do decent back-of-the-envelope calculations!

Encoding Your Data

- CPUs are fast, memory/bandwidth are precious, ergo…
  - Variable-length encodings
  - Compression
  - Compact in-memory representations

- Compression/encoding very important for many systems
  - inverted index posting list formats
  - storage systems for persistent data

- We have lots of core libraries in this area
  - Many tradeoffs: space, encoding/decoding speed, etc. E.g.:
    - Zippy: encode@300 MB/s, decode@600MB/s, 2-4X compression
    - gzip: encode@25MB/s, decode@200MB/s, 4-6X compression

Designing & Building Infrastructure

Identify common problems, and build software systems to address them in a general way

- Important not to try to be all things to all people
  - Clients might be demanding 8 different things
  - Doing 6 of them is easy
  - …handling 7 of them requires real thought
  - …dealing with all 8 usually results in a worse system
    - more complex, compromises other clients in trying to satisfy everyone

Don't build infrastructure just for its own sake

- Identify common needs and address them
- Don't imagine unlikely potential needs that aren't really there
- Best approach: use your own infrastructure (especially at first!)
  - (much more rapid feedback about what works, what doesn't)

Design for Growth

Try to anticipate how requirements will evolve

- keep likely features in mind as you design base system

Ensure your design works if scale changes by 10X or 20X

- but the right solution for X often not optimal for 100X

Interactive Apps: Design for Low Latency

- Aim for low avg. times (happy users!)
  - 90%ile and 99%ile also very important
  - Think about how much data you're shuffling around
    - e.g. dozens of 1 MB RPCs per user request -> latency will be lousy

- Worry about variance!
  - Redundancy or timeouts can help bring in latency tail
- Judicious use of caching can help
- Use higher priorities for interactive requests
- Parallelism helps!

Making Applications Robust Against Failures

Canary requests

Failover to other replicas/datacenters

Bad backend detection:

- stop using for live requests until behavior gets better

More aggressive load balancing when imbalance is more severe

Make your apps do something reasonable even if not all is right

- Better to give users limited functionality than an error page

Add Sufficient Monitoring/Status/Debugging Hooks

All our servers:

- Export HTML-based status pages for easy diagnosis
- Export a collection of key-value pairs via a standard interface
  - monitoring systems periodically collect this from running servers
- RPC subsystem collects sample of all requests, all error requests, all requests >0.0s, >0.05s, >0.1s, >0.5s, >1s, etc.

- Support low-overhead online profiling
  - cpu profiling
  - memory profiling
  - lock contention profiling

If your system is slow or misbehaving, can you figure out why?

MapReduce

- A simple programming model that applies to many large-scale computing problems

- Hide messy details in MapReduce runtime library:
  - automatic parallelization
  - load balancing
  - network and disk transfer optimizations
  - handling of machine failures
  - robustness
  - **improvements to core library benefit all users of library!**

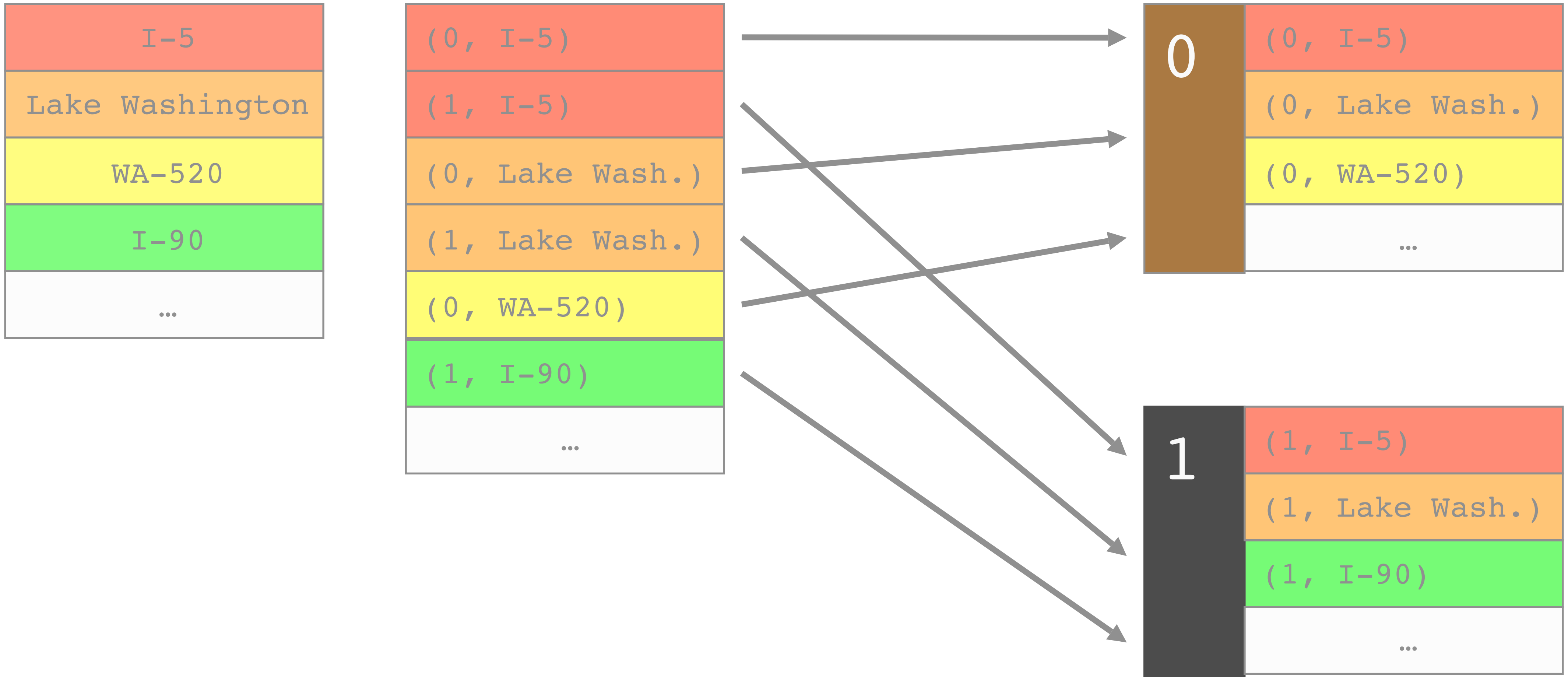
Typical problem solved by MapReduce

- Read a lot of data
- Map: extract something you care about from each record
- Shuffle and Sort
- Reduce: aggregate, summarize, filter, or transform
- Write the results

Outline stays the same,
map and reduce change to fit the problem

Example: Rendering Map Tiles

Input	Map	Shuffle	Reduce	Output
Geographic feature list	Emit each to all overlapping latitude-longitude rectangles	Sort by key (key= Rect. Id)	Render tile using data for all enclosed features	Rendered tiles



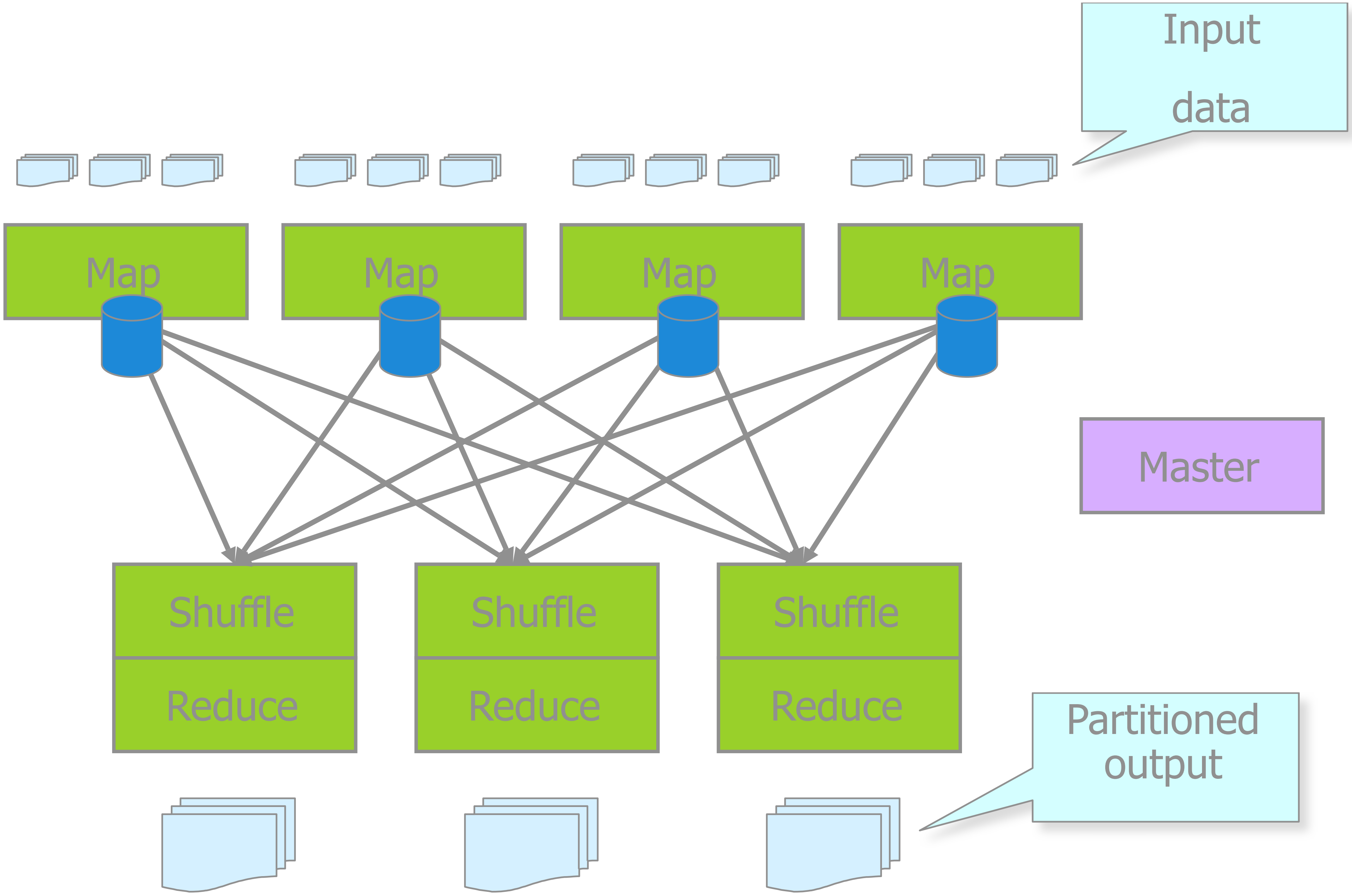
- I-5
- Lake Washington
- WA-520
- I-90
- …

- (0, I-5)
- (1, I-5)
- (0, Lake Wash.)
- (1, Lake Wash.)
- (0, WA-520)
- (1, I-90)
- …

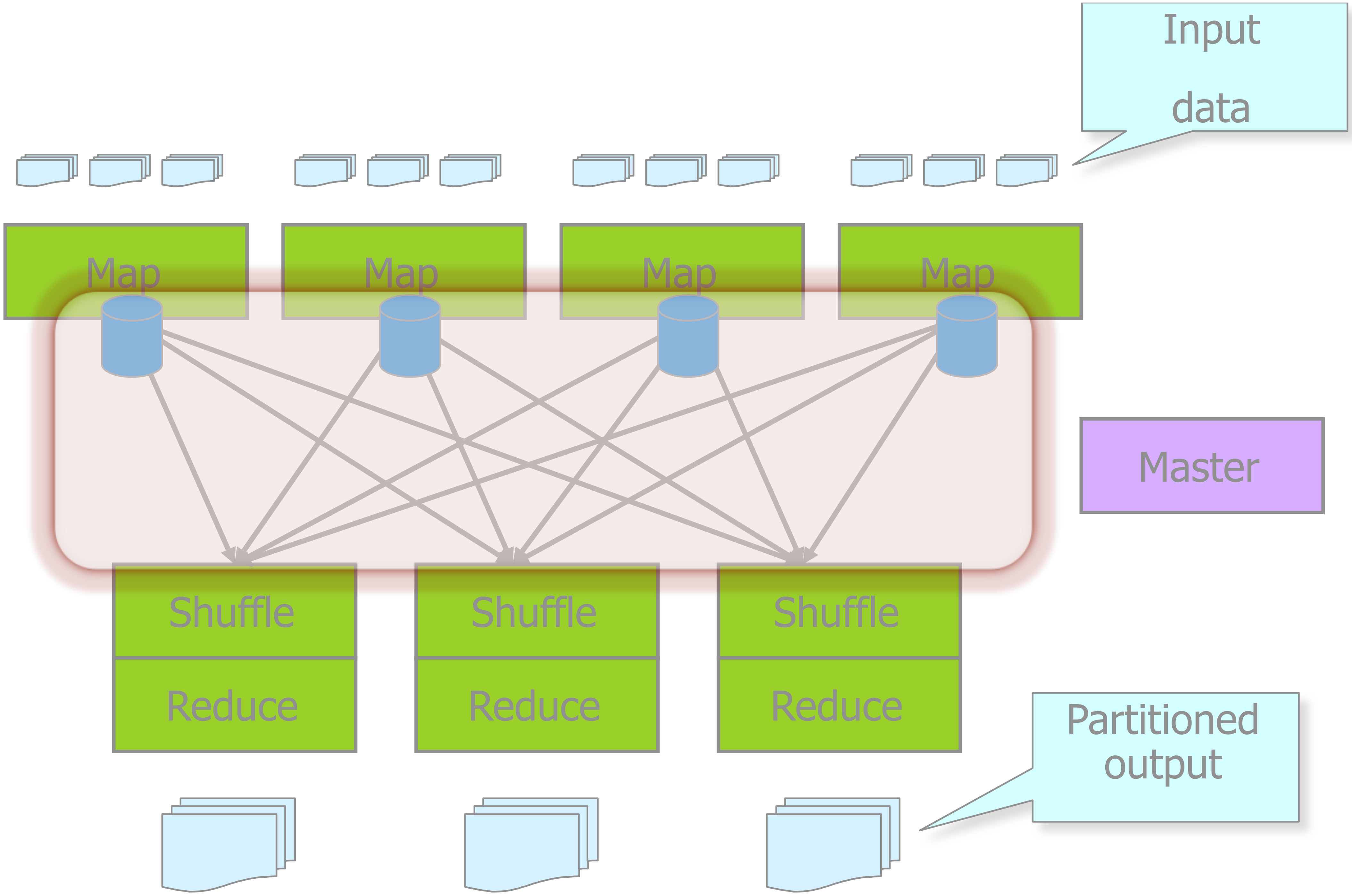
0
- (0, I-5)
- (0, Lake Wash.)
- (0, WA-520)
- …

1
- (1, I-5)
- (1, Lake Wash.)
- (1, I-90)
- …

Parallel MapReduce



Parallel MapReduce



For large enough problems, it's more about disk and network performance than CPU & DRAM

MapReduce Usage Statistics Over Time

	Aug, ‘04	Mar, ‘06	Sep, '07	Sep, ’09
Number of jobs	29K	171K	2,217K	3,467K
Average completion time (secs)	634	874	395	475
Machine years used	217	2,002	11,081	25,562
Input data read (TB)	3,288	52,254	403,152	544,130
Intermediate data (TB)	758	6,743	34,774	90,120
Output data written (TB)	193	2,970	14,018	57,520
Average worker machines	157	268	394	488

MapReduce in Practice

- Abstract input and output interfaces
  - lots of MR operations don't just read/write simple files
    - B-tree files
    - memory-mapped key-value stores
    - complex inverted index file formats
    - BigTable tables
    - SQL databases, etc.
    - ...

- Low-level MR interfaces are in terms of byte arrays
  - Hardly ever use textual formats, though: slow, hard to parse
  - Most input & output is in encoded Protocol Buffer format

- See "*MapReduce: A Flexible Data Processing Tool*" (to appear in upcoming CACM)

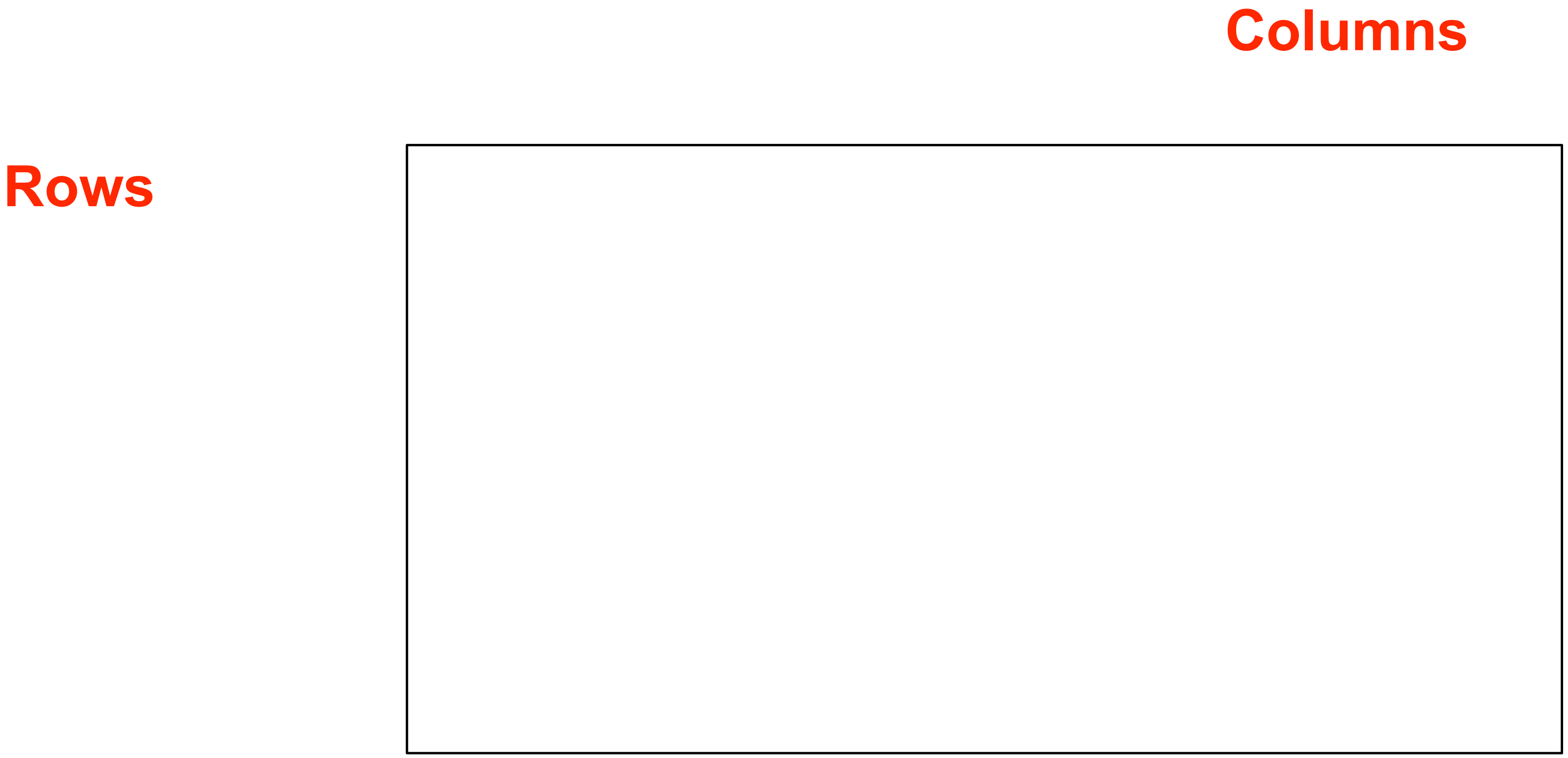
BigTable: Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Scale is large
  - billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands of q/sec
  - 100TB+ of satellite image data

Basic Data Model

- Distributed multi-dimensional sparse map

(row, column, timestamp) → cell contents

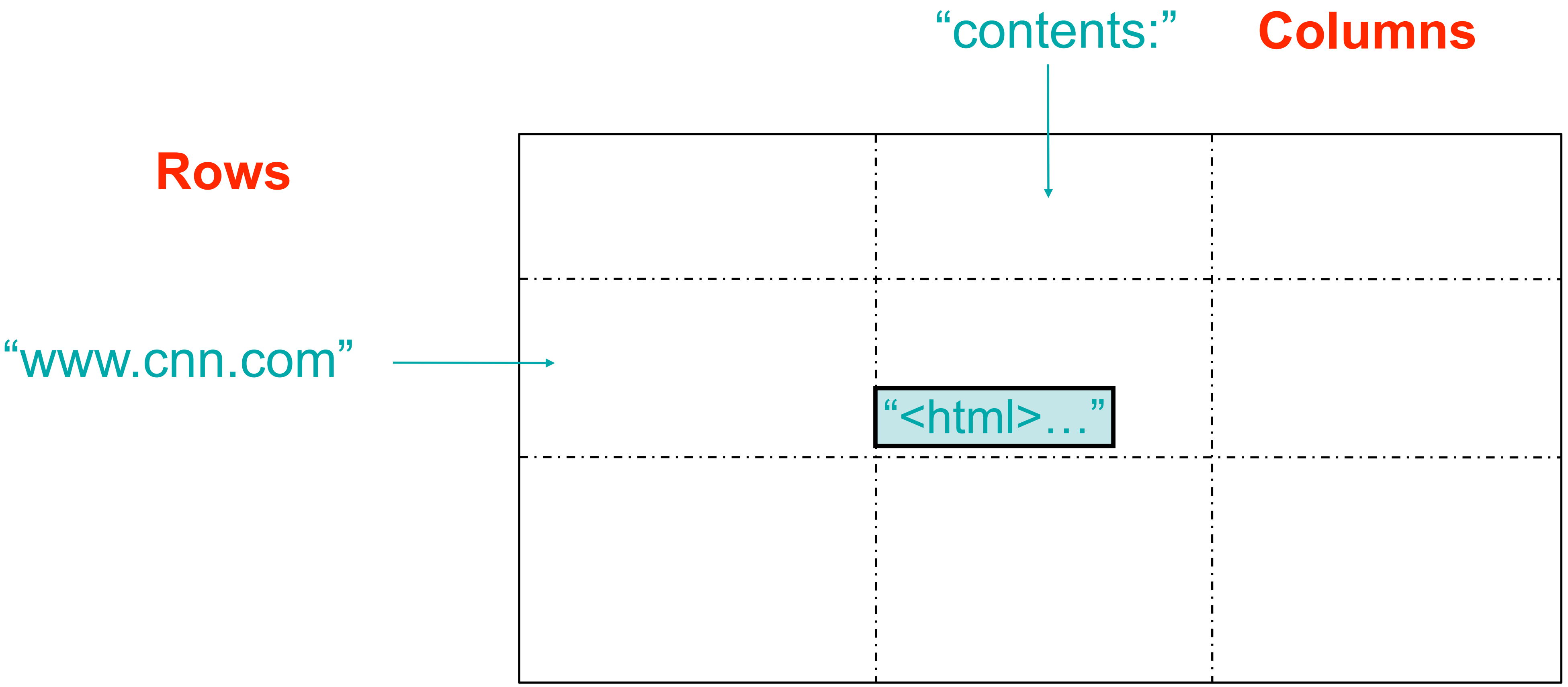


- Rows are ordered lexicographically
- Good match for most of our applications

Basic Data Model

- Distributed multi-dimensional sparse map

  (row, column, timestamp) → cell contents

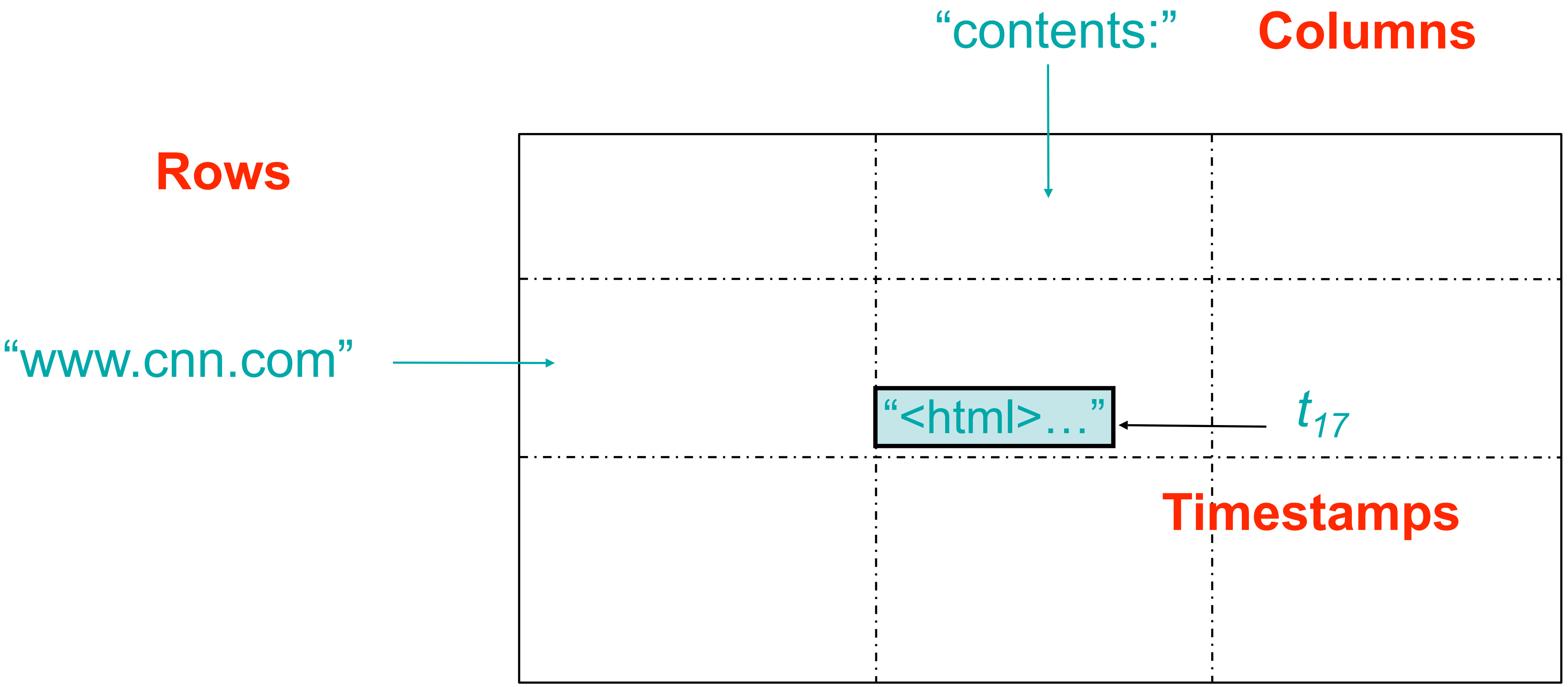


- Rows are ordered lexicographically
- Good match for most of our applications

Basic Data Model

- Distributed multi-dimensional sparse map

  (row, column, timestamp) → cell contents

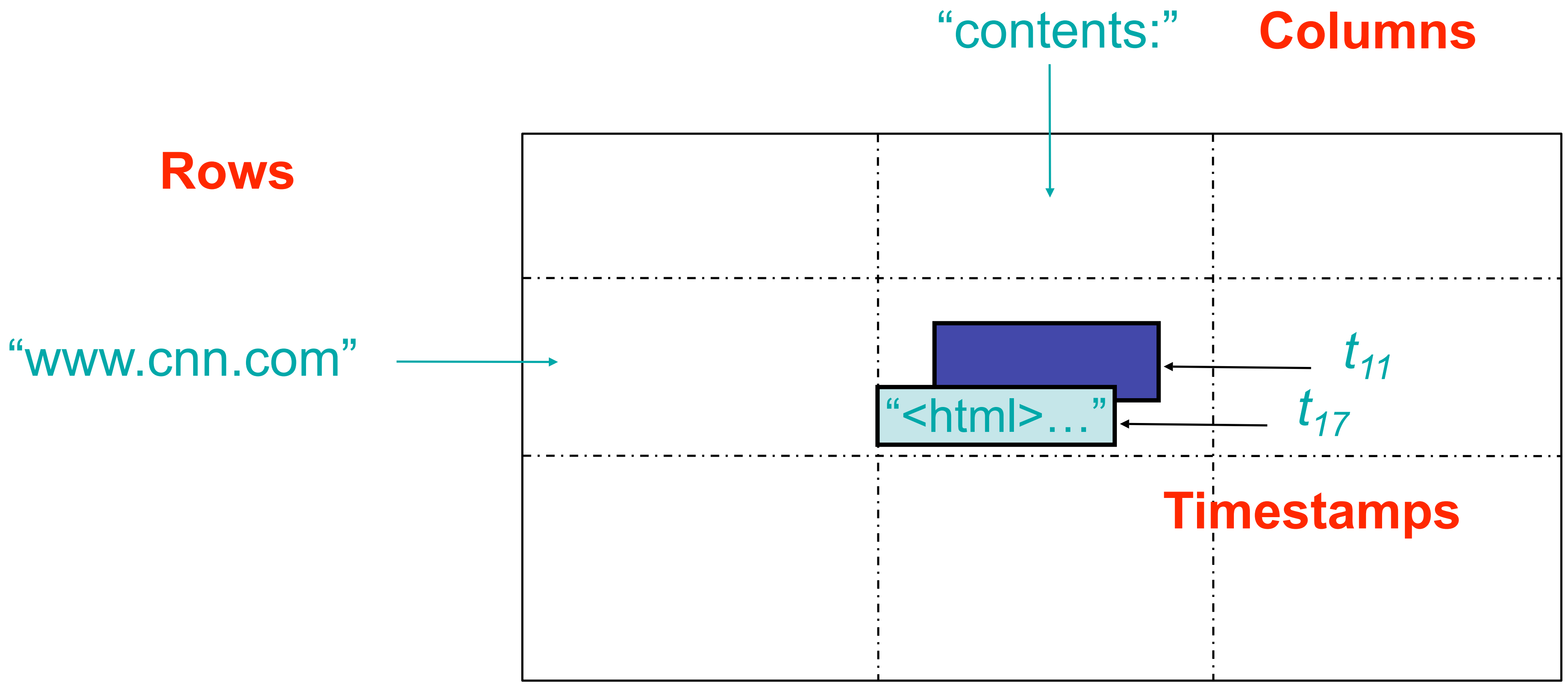


- Rows are ordered lexicographically
- Good match for most of our applications

Basic Data Model

- Distributed multi-dimensional sparse map

  (row, column, timestamp) → cell contents



"contents:"

Columns

Rows

"www.cnn.com"

"<html>…"

t_{11}

t_{17}

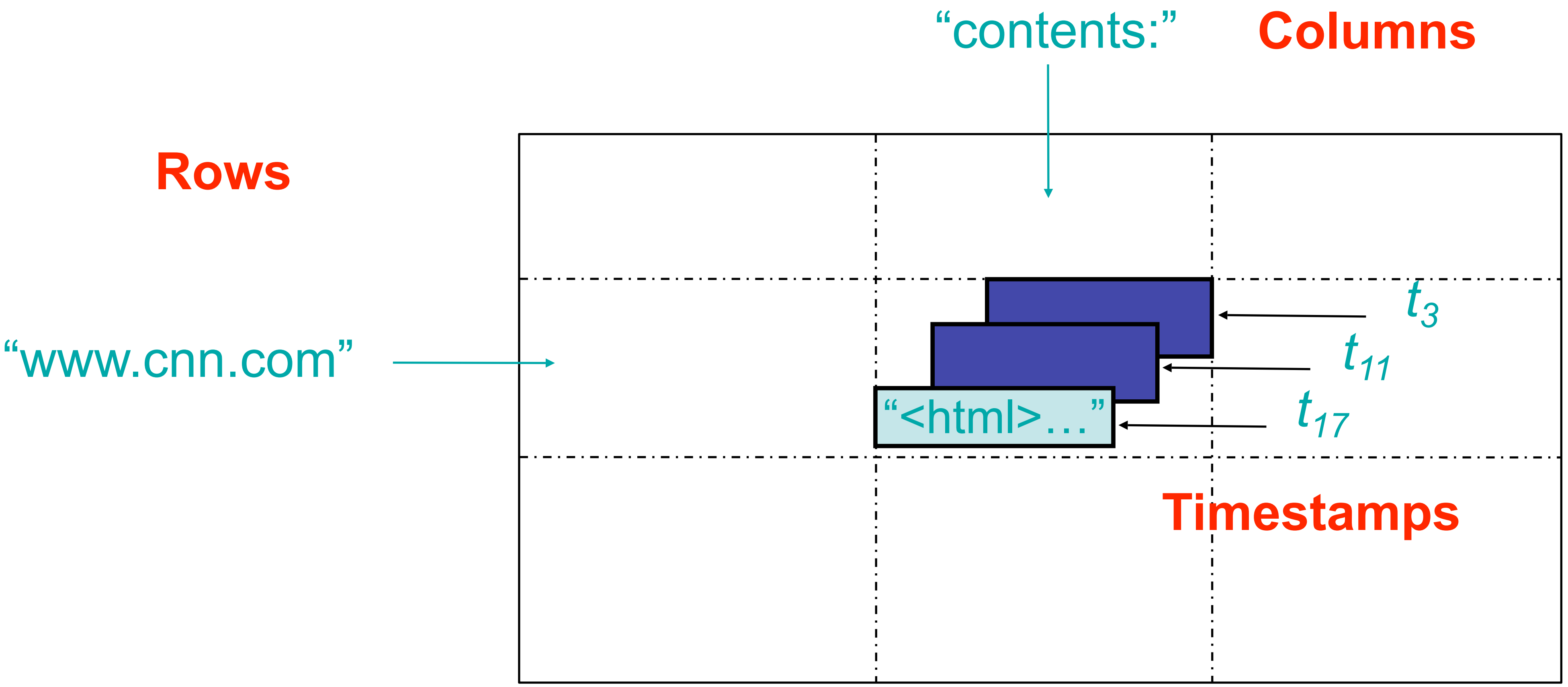
Timestamps

- Rows are ordered lexicographically
- Good match for most of our applications

Basic Data Model

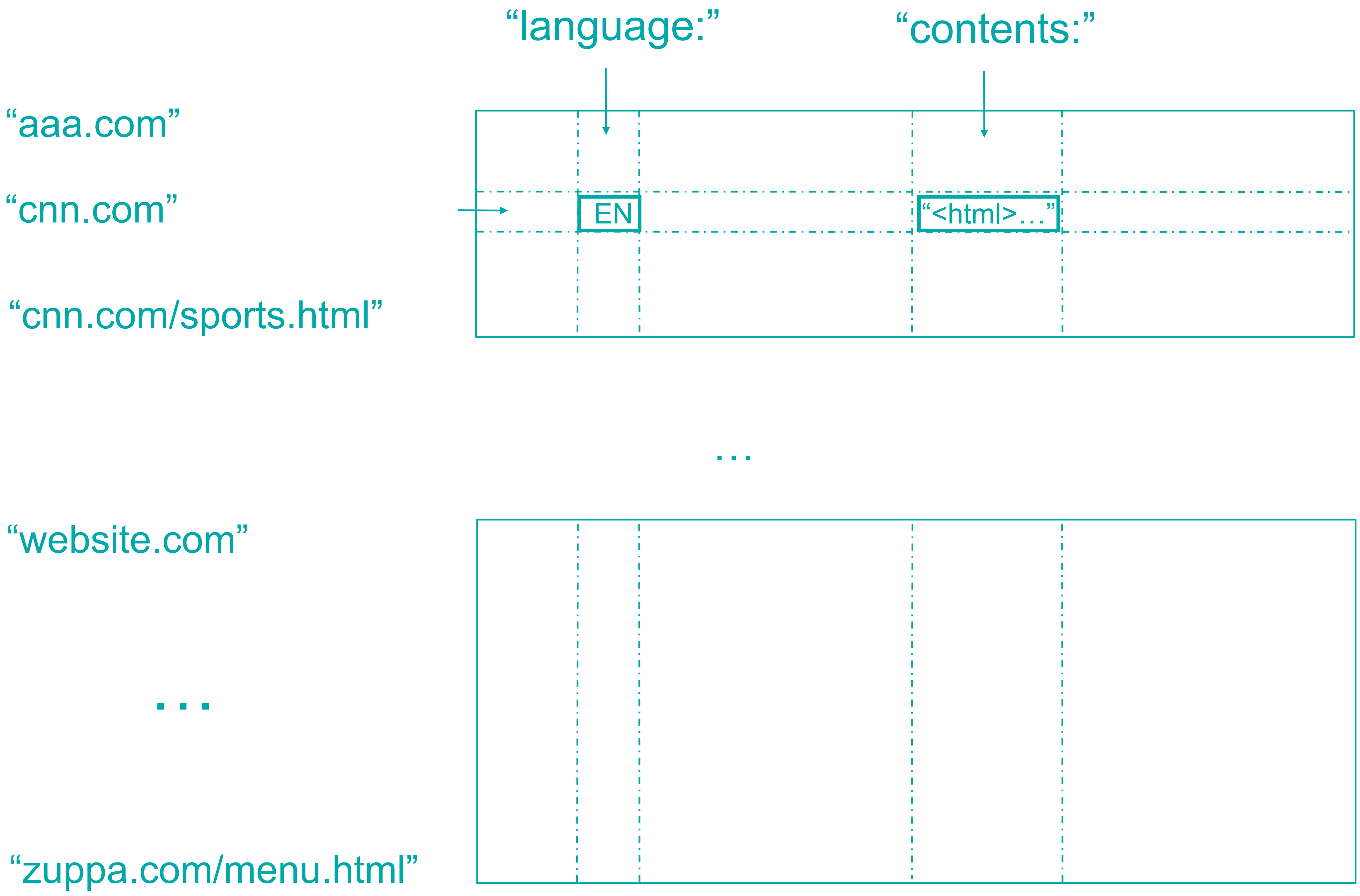
- Distributed multi-dimensional sparse map

  (row, column, timestamp) → cell contents



- Rows are ordered lexicographically
- Good match for most of our applications

Tablets & Splitting



“language:”

“contents:”

“aaa.com”

“cnn.com”

EN

“<html>…”

“cnn.com/sports.html”

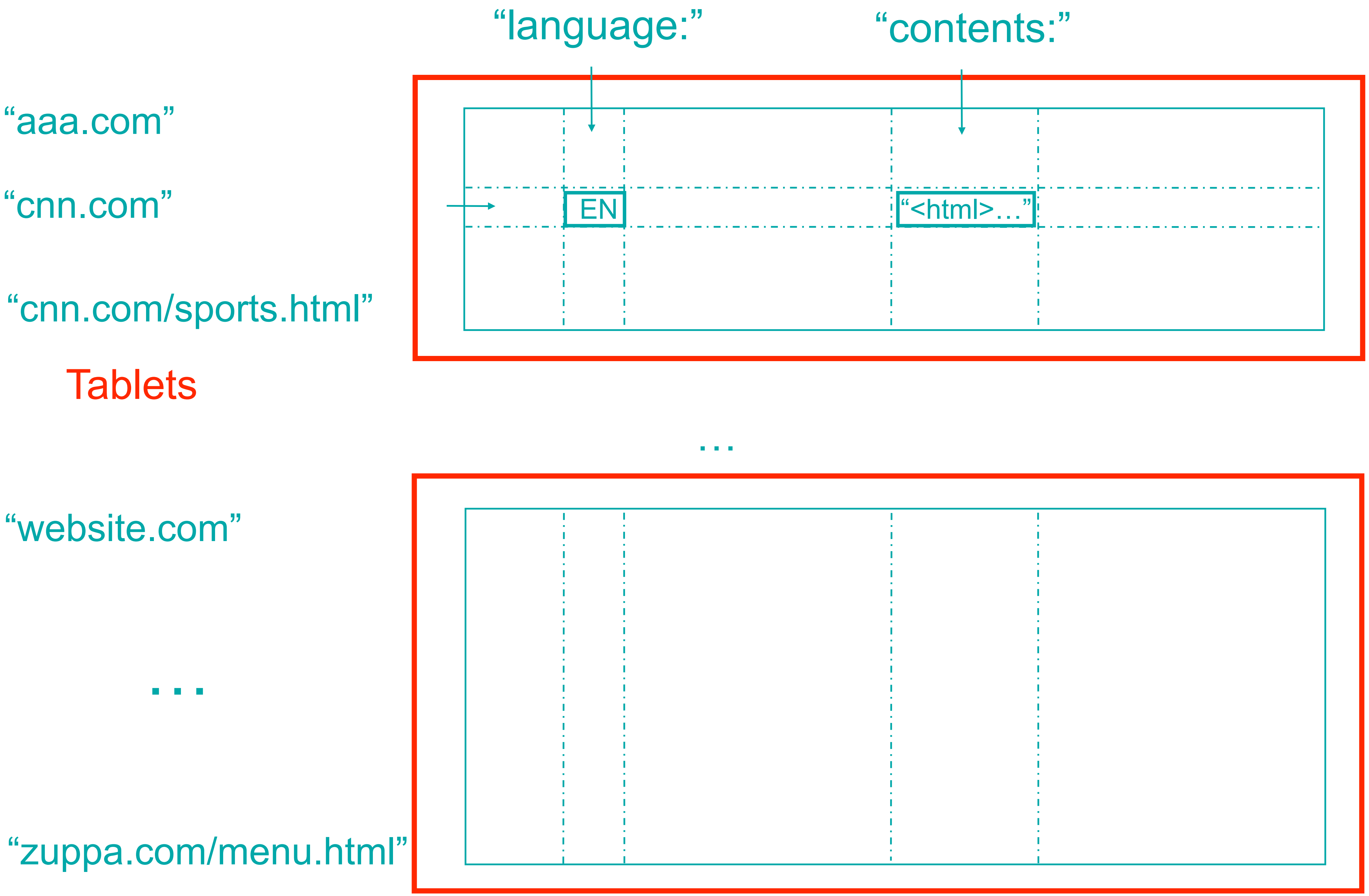
…

“website.com”

…

“zuppa.com/menu.html”

Tablets & Splitting



"language:"

"contents:"

"aaa.com"

"cnn.com"

EN

"<html>…"

"cnn.com/sports.html"

Tablets

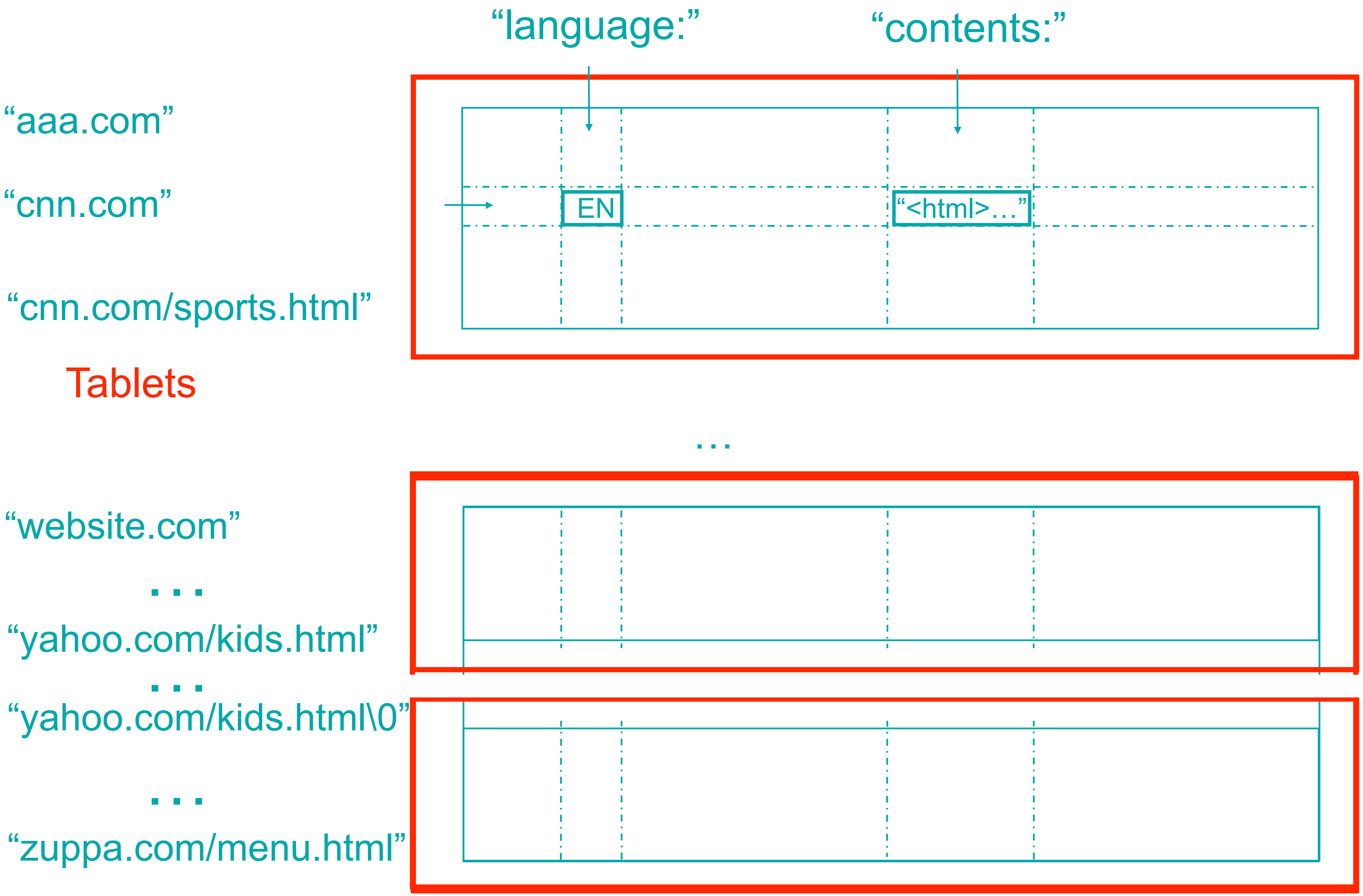
…

"website.com"

…

"zuppa.com/menu.html"

Tablets & Splitting



“language:”

“contents:”

“aaa.com”

“cnn.com”

EN

“<html>…”

“cnn.com/sports.html”

Tablets

…

“website.com”

. . .

“yahoo.com/kids.html”

. . .

“yahoo.com/kids.html\0”

. . .

“zuppa.com/menu.html”

BigTable System Structure

Bigtable Cell

Bigtable master

Bigtable tablet server

Bigtable tablet server

…

Bigtable tablet server

BigTable System Structure

Bigtable Cell

Bigtable master

performs metadata ops +
load balancing

Bigtable tablet server

Bigtable tablet server

…

Bigtable tablet server

BigTable System Structure

Bigtable Cell

Bigtable master

performs metadata ops +
load balancing

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

BigTable System Structure

Bigtable Cell

Bigtable master

performs metadata ops +
load balancing

Bigtable tablet server — serves data

Bigtable tablet server — serves data

…

Bigtable tablet server — serves data

Cluster scheduling system

GFS

Lock service

BigTable System Structure

Bigtable Cell

Bigtable master

performs metadata ops +
load balancing

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

Cluster scheduling system

handles failover, monitoring

GFS

Lock service

BigTable System Structure

Bigtable Cell

Bigtable master

performs metadata ops +
load balancing

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

Cluster scheduling system

handles failover, monitoring

GFS

holds tablet data, logs

Lock service

BigTable System Structure

Bigtable Cell

Bigtable master

performs metadata ops +
load balancing

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

Cluster scheduling system

handles failover, monitoring

GFS

holds tablet data, logs

Lock service

holds metadata,
handles master-election

BigTable System Structure

Bigtable Cell

Bigtable client

Bigtable client library

Bigtable master

performs metadata ops + load balancing

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

Cluster scheduling system

handles failover, monitoring

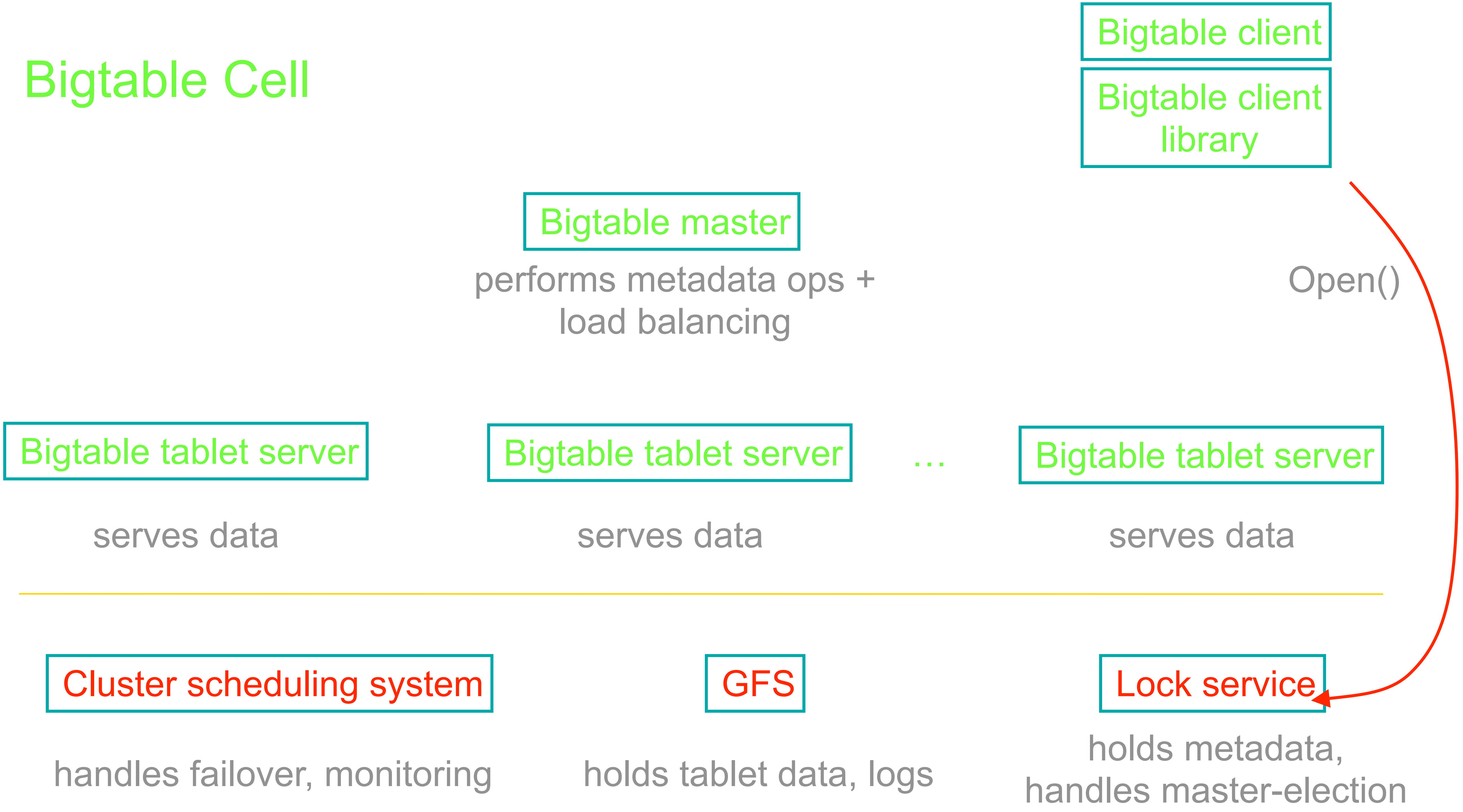
GFS

holds tablet data, logs

Lock service

holds metadata, handles master-election

BigTable System Structure



Bigtable Cell

Bigtable client

Bigtable client library

Open()

Bigtable master

performs metadata ops + load balancing

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

Cluster scheduling system

handles failover, monitoring

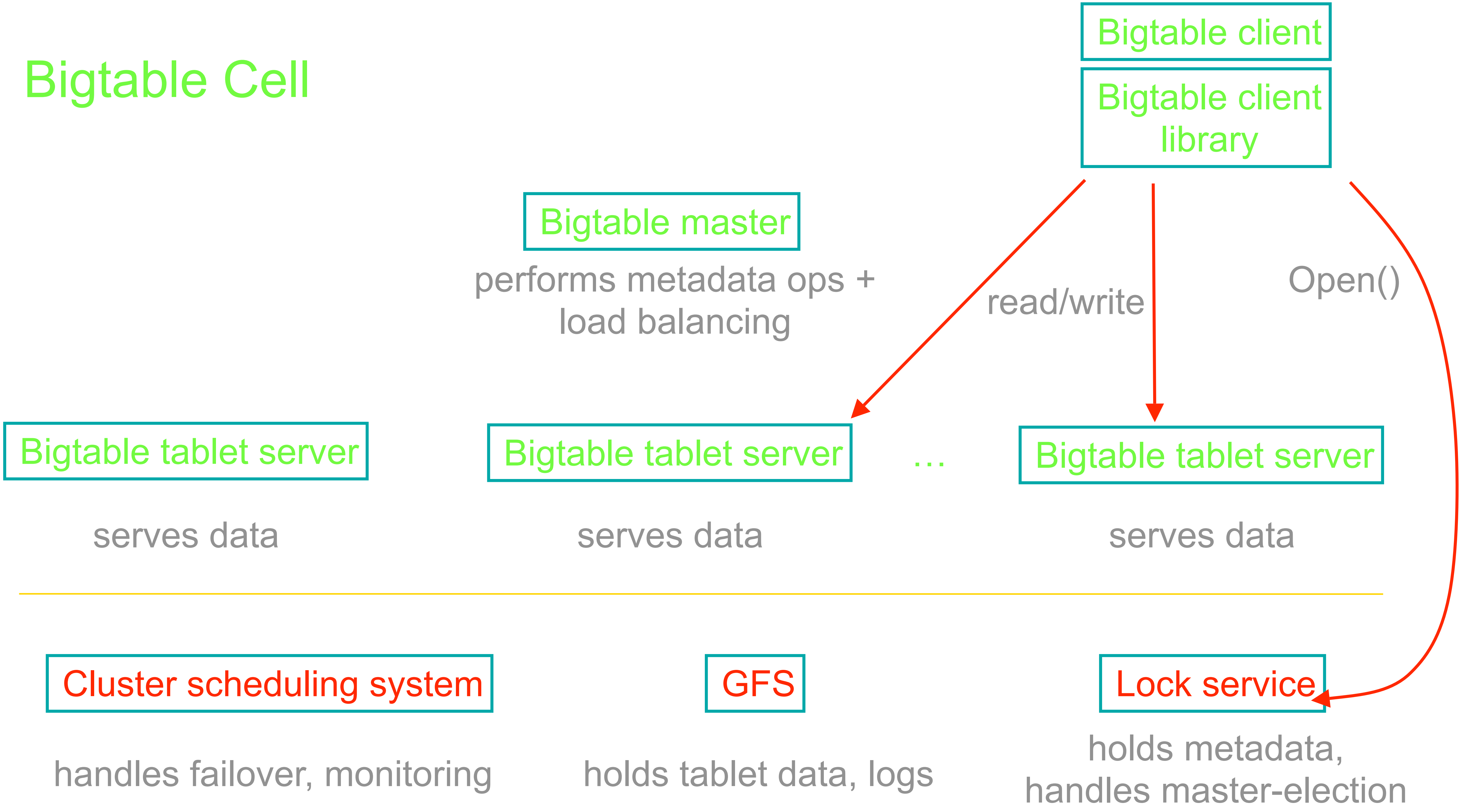
GFS

holds tablet data, logs

Lock service

holds metadata, handles master-election

BigTable System Structure



Bigtable Cell

Bigtable client

Bigtable client library

Bigtable master

performs metadata ops + load balancing

read/write

Open()

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

Cluster scheduling system

handles failover, monitoring

GFS

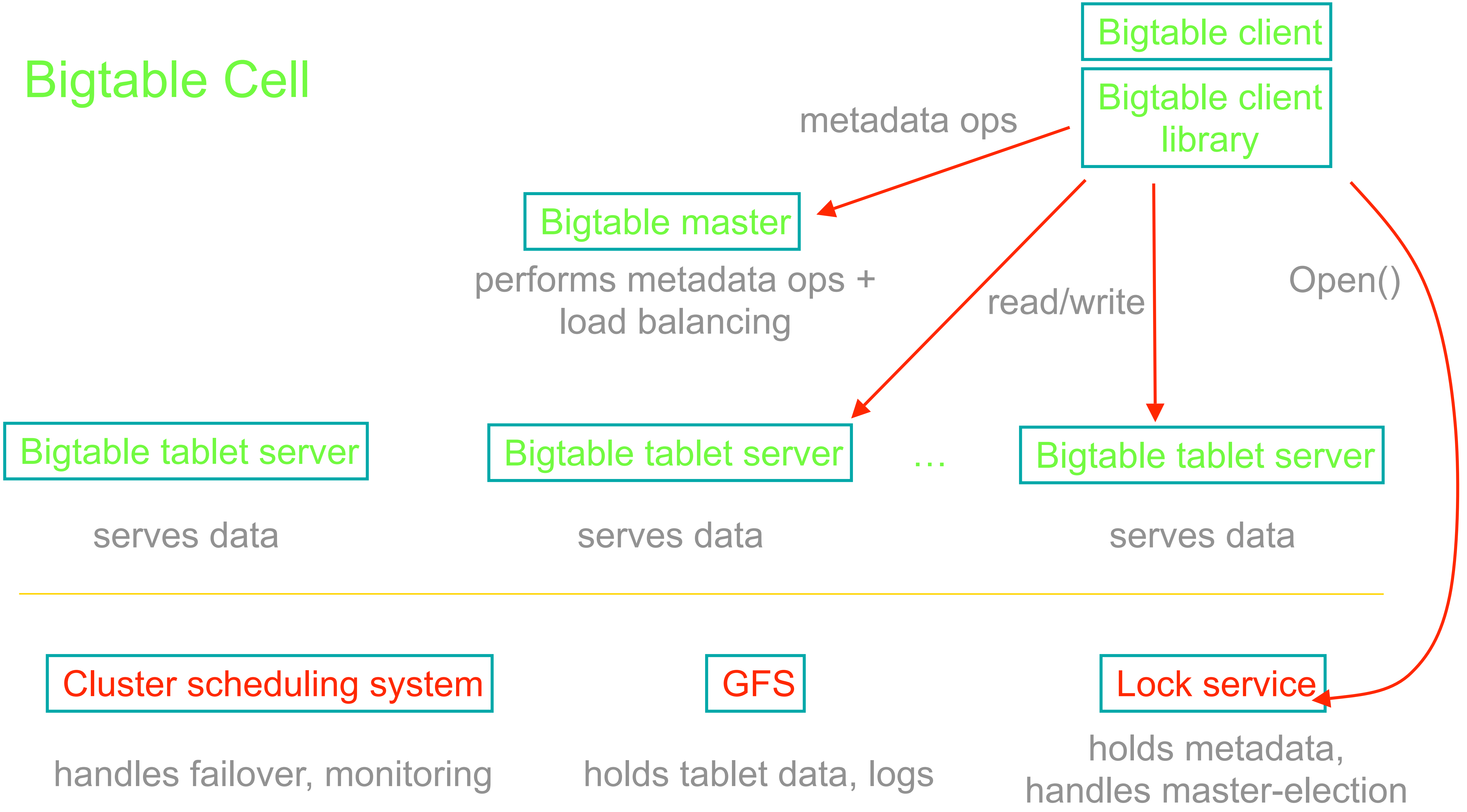
holds tablet data, logs

Lock service

holds metadata, handles master-election

Google

BigTable System Structure



Bigtable Cell

Bigtable client

Bigtable client library

metadata ops

Bigtable master

performs metadata ops + load balancing

read/write

Open()

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

Cluster scheduling system

handles failover, monitoring

GFS

holds tablet data, logs

Lock service

holds metadata, handles master-election

BigTable Status

- Design/initial implementation started beginning of 2004
- Production use or active development for 100+ projects:
  - Google Print
  - My Search History
  - Orkut
  - Crawling/indexing pipeline
  - Google Maps/Google Earth
  - Blogger
  - …
- Currently ~500 BigTable clusters
- Largest cluster:
  - 70+ PB data; sustained: 10M ops/sec; 30+ GB/s I/O

BigTable: What's New Since OSDI'06?

- Lots of work on scaling
- Service clusters, managed by dedicated team
- Improved performance isolation
  - fair-share scheduler within each server, better accounting of memory used per user (caches, etc.)
  - can partition servers within a cluster for different users or tables

- Improved protection against corruption
  - many small changes
  - e.g. immediately read results of every compaction, compare with CRC.
    - Catches ~1 corruption/5.4 PB of data compacted

BigTable Replication (New Since OSDI'06)

- Configured on a per-table basis
- Typically used to replicate data to multiple bigtable clusters in different data centers
- *Eventual consistency model*: writes to table in one cluster eventually appear in all configured replicas
- Nearly all user-facing production uses of BigTable use replication

BigTable Coprocessors (New Since OSDI'06)

- Arbitrary code that runs run next to each tablet in table
  - as tablets split and move, coprocessor code automatically splits/moves too

- High-level call interface for clients
  - Unlike RPC, calls addressed to rows or ranges of rows
    - coprocessor client library resolves to actual locations
  - Calls across multiple rows automatically split into multiple parallelized RPCs

- Very flexible model for building distributed services
  - automatic scaling, load balancing, request routing for apps

Example Coprocessor Uses

- Scalable metadata management for Colossus (next gen GFS-like file system)

- Distributed language model serving for machine translation system

- Distributed query processing for full-text indexing support

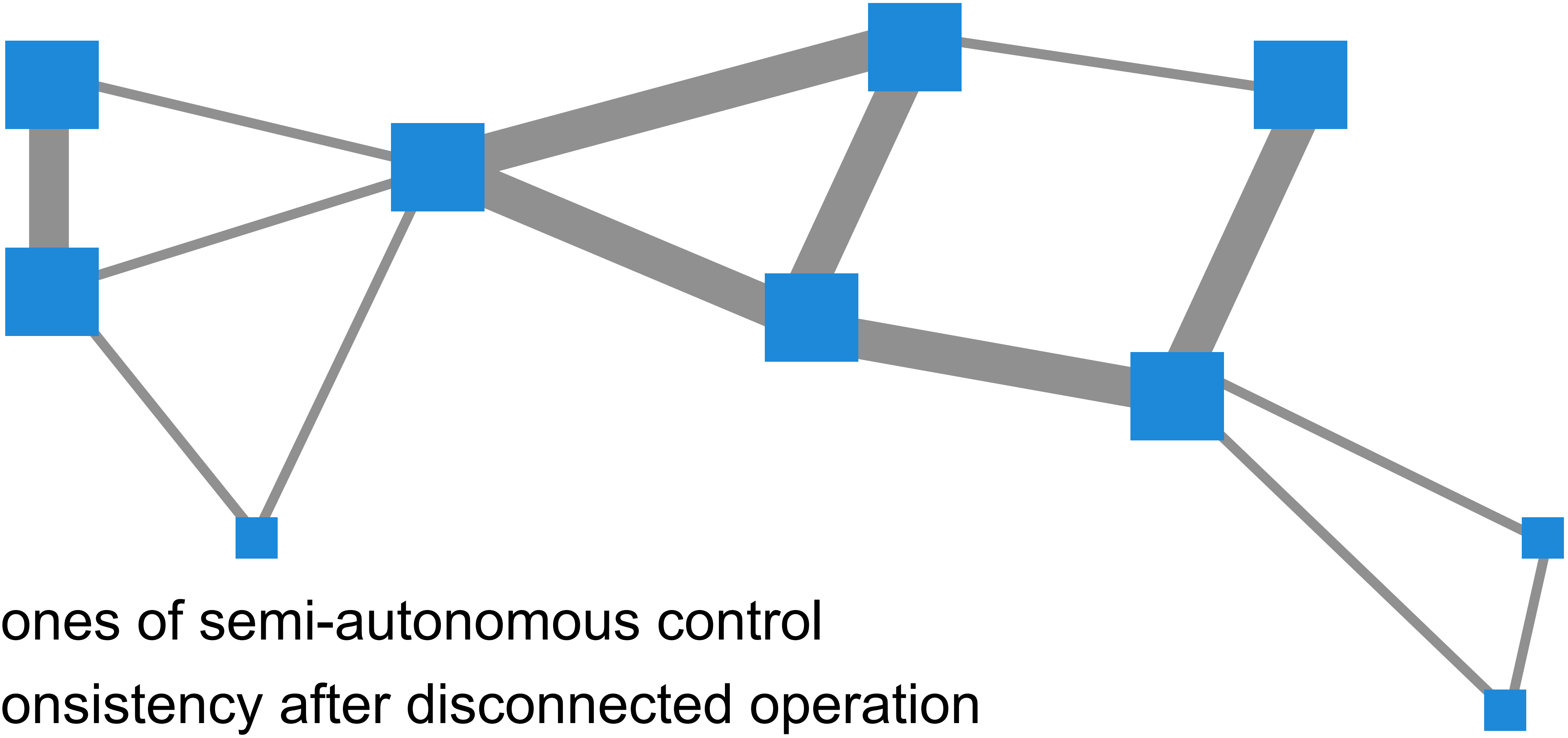
- Regular expression search support for code repository

Current Work: Spanner

- Storage & computation system that spans all our datacenters
  - single global namespace
    - Names are independent of location(s) of data
    - Similarities to Bigtable: tables, families, locality groups, coprocessors, ...
    - Differences: hierarchical directories instead of rows, fine-grained replication
    - Fine-grained ACLs, replication configuration at the per-directory level
  - support mix of strong and weak consistency across datacenters
    - Strong consistency implemented with Paxos across tablet replicas
    - Full support for distributed transactions across directories/machines
  - much more automated operation
    - system automatically moves and adds replicas of data and computation based on constraints and usage patterns
    - automated allocation of resources across entire fleet of machines

Design Goals for Spanner

- Future scale: ~10^6 to 10^7 machines, ~10^{13} directories, ~10^{18} bytes of storage, spread at 100s to 1000s of locations around the world, ~10^9 client machines



  - zones of semi-autonomous control
  - consistency after disconnected operation
  - users specify high-level desires:

    "99%ile latency for accessing this data should be <50ms"

    "Store this data on at least 2 disks in EU, 2 in U.S. & 1 in Asia"

Adaptivity in World-Wide Systems

- Challenge: automatic, dynamic world-wide placement of data & computation to minimize latency and/or cost, given constraints on:
  - bandwidth
  - packet loss
  - power
  - resource usage
  - failure modes
  - ...
- Users specify high-level desires:

  “99%ile latency for accessing this data should be <50ms”

  “Store this data on at least 2 disks in EU, 2 in U.S. & 1 in Asia”

Building Applications on top of Weakly Consistent Storage Systems

- Many applications need state replicated across a wide area
  - For reliability and availability

- Two main choices:
  - consistent operations (e.g. use Paxos)
    - often imposes additional latency for common case
  - inconsistent operations
    - better performance/availability, but apps harder to write and reason about in this model

- Many apps need to use a mix of both of these:
  - e.g. Gmail: marking a message as read is asynchronous, sending a message is a heavier-weight consistent operation

Building Applications on top of Weakly Consistent Storage Systems

- Challenge: General model of consistency choices, explained and codified
  - ideally would have one or more “knobs” controlling performance vs. consistency
  - “knob” would provide easy-to-understand tradeoffs

- Challenge: Easy-to-use abstractions for resolving conflicting updates to multiple versions of a piece of state
  - Useful for reconciling client state with servers after disconnected operation
  - Also useful for reconciling replicated state in different data centers after repairing a network partition

Thanks! Questions...?

Further reading:

- Ghemawat, Gobioff, & Leung. *Google File System*, SOSP 2003.
- Barroso, Dean, & Hölzle . *Web Search for a Planet: The Google Cluster Architecture*, IEEE Micro, 2003.
- Dean & Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004.
- Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, & Gruber. *Bigtable: A Distributed Storage System for Structured Data*, OSDI 2006.
- Burrows. *The Chubby Lock Service for Loosely-Coupled Distributed Systems*. OSDI 2006.
- Pinheiro, Weber, & Barroso. *Failure Trends in a Large Disk Drive Population*. FAST 2007.
- Brants, Popat, Xu, Och, & Dean. *Large Language Models in Machine Translation*, EMNLP 2007.
- Barroso & Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan & Claypool Synthesis Series on Computer Architecture, 2009.
- Malewicz *et al*. *Pregel: A System for Large-Scale Graph Processing*. PODC, 2009.
- Schroeder, Pinheiro, & Weber. *DRAM Errors in the Wild: A Large-Scale Field Study*. SEGMETRICS’09.
- Protocol Buffers. http://code.google.com/p/protobuf/

These and many more available at: http://labs.google.com/papers.html